Speeding:
On Low-Latency Key Exchange

Britta Hale[1] and Tibor Jager[2] and Sebastian Lauer[2] and Jörg Schwenk[2]

[1] Horst Görtz Institute, Ruhr-University Bochum
`{tibor.jager, sebastian.lauer, joerg.schwenk}@rub.de`
[2] Norwegian University of Science and Technology, NTNU, Trondheim
`britta.hale@item.ntnu.no`

Abstract. Low-latency key exchange (LLKE) protocols allow for the transmission of cryptographically protected payload data without requiring the prior exchange of messages of a cryptographic key exchange protocol, while providing perfect forward secrecy. The LLKE concept was first realized by Google in the QUIC protocol, and a low-latency mode is currently under discussion for inclusion in TLS 1.3.

In LLKE two keys are generated, typically using a Diffie-Hellman key exchange. The first key is a combination of an ephemeral client share and a long-lived server share. The second key is computed using an ephemeral server share and the same ephemeral client share.

In this paper, we propose (relatively) simple, novel security models, which catch the intuition behind known LLKE protocols; namely that the first (respectively, second) key should remain indistinguishable from a random value, even if the second (respectively, first) key is revealed. We call this property *strong key independence*. We also give the first constructions of LLKE which are provably secure in these models, based on the generic assumption that secure non-interactive key exchange (NIKE) exists.

Keywords: Foundations, low-latency key exchange, zero-RTT protocols, authenticated key exchange, non-interactive key exchange, QUIC, TLS 1.3.

1 Introduction

Ultimately, the first generation of internet key exchange protocols did not care too much about efficiency (in terms of messages to be exchanged before a key is established), since secure connections were considered to be the exception rather than the rule: SSL (versions 2.0 and 3.0) and TLS (versions 1.0, 1.1, and 1.2) require 2 round-trip times (RTT) for key establishment before the first cryptographically-protected payload data can be sent. With the increased use encryption,[3] efficiency becomes a more and more important aspect for protocols like TLS. Similarly, the older IPSec IKE version v1 needs between 3 RTT (aggressive mode + quick mode) and 4.5 RTT (main mode + quick mode). This was soon realized to be problematic, and in IKEv2 the number of RTTs was reduced to 2.

---

[3] Think of initiatives like Let's Encrypt (`https://letsencrypt.org/`), for example.

The QUIC protocol. Fundamentally, the discussion on low-latency key exchange (LLKE, aka. zero-RTT or 0-RTT key exchange) was opened when Google proposed the QUIC protocol.[4] QUIC (cf. Figure 1) achieves low-latency by caching a signed server configuration file on the client side, which contains a medium-lived Diffie-Hellman (DH) share $Y_0 = g^{y_0}$.[5]

When a client wishes to establish a connection with a server and possesses a valid configuration file of that server, it chooses a fresh ephemeral DH share $X = g^x$ and computes a temporal key k_1 from $g^{y_0 x}$. Using this key k_1, the client can encrypt and authenticate data to be sent to the server, together with X. In response, the server sends a fresh DH share $Y = g^y$ and computes a session key k_2 from g^{xy}, which is used for all subsequent data exchanges.

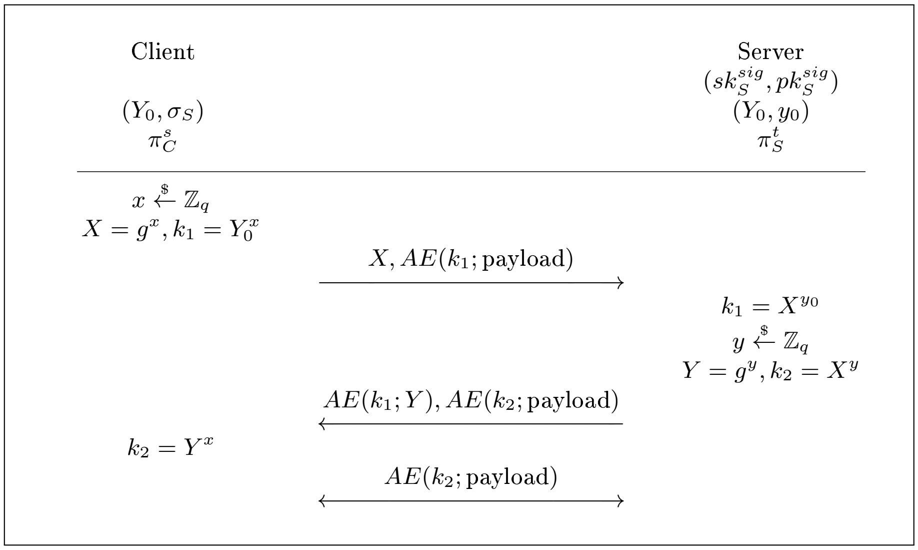

Fig. 1: Google's QUIC protocol (simplified) with cached server key configuration file (Y_0, σ_S). AE denotes a symmetric authenticated encryption algorithm (e.g., AES-GCM).

TLS 1.3. The current TLS 1.3 draft `draft-ietf-tls-tls13-08` [19] contains a 0-RTT exchange mode where a `ServerConfiguration` message is cached by the client. In contrast to QUIC, this message is not signed, but must instead be communicated to the client during a valid "normal" handshake (including a signature by the server). This `ServerConfiguration` message contains a semi-static server DH (or Elliptic Curve DH) share $Y_0 = g^{y_0}$ which essentially plays

---

[4] See `https://www.chromium.org/quic`

[5] If the client does not have a valid file, it has to be requested from the server, which increases the number of RTTs by 1, but may then be re-used for future sessions.

the same role as in QUIC. Unlike QUIC, however, TLS 1.3 uses digital signatures to authenticate the server.

When the client sends its DH share $X = g^x$ in the `ClientKeyShare` message, he may already encrypt additional handshake extensions, the client's (optional) certificate and signature, and even application data, with a key k_1 derived from $Y_0^x = g^{y_0 x}$. Thus data can be encrypted with 0-RTT overhead.

Security goals. LLKE protocols like QUIC and TLS 1.3 have ad-hoc designs that should achieve three security goals: (1) 0-RTT encryption, where ciphertext data can already be sent together with the first handshake message; (2) perfect forward secrecy (PFS), where all ciphertexts exchanged after the second handshake message will remain secure even after the (static or semi-static) private keys of the server have been leaked, and (3) key independence, where "knowledge" about one of the two symmetric keys generated should not endanger the "security" of the other key.

Strong key independence. Intuitively, an LLKE protocol should achieve *strong key independence* between k_1 and k_2; if any one of the two keys is leaked at any time, the other key should still be indistinguishable from a random value. In all known security models, this intuition would be formalized as follows: if the adversary $\mathcal{A}$ asks a Reveal query for k_1, he is still allowed to ask a Test query for k_2, and vice versa. If the two keys are computationally independent from each other (which also includes computations on the different protocol messages), then the adversary should have only a negligible advantage in answering the Test query correctly.

The research questions to be answered are the following: *Do existing examples of 0-RTT protocols have strong key independence? Can we describe a generic way to construct LLKE protocols that provably achieve strong key independence?*

QUIC does not provide strong key independence. If an attacker $\mathcal{A}$ is allowed to learn k_1 by a Reveal-query, then he is able to decrypt $AE(k_1;Y)$ and re-encrypt its own value $Y^* := g^{y^*}$ as $AE(k_1;Y^*)$. Then he can compute the same $k_2 = X^{y^*}$ as the client oracle, and can thus distinguish between the "real" key and a "random" key chosen by the Test query.

Note that this theoretical attack does not imply that QUIC is insecure. It only shows that the authenticity of the server's Diffie-Hellman share, which is sent in QUIC to establish k_2, depends strongly on the security of key k_1. Therefore QUIC does not provide strong key independence in the sense sketched above.

The case of TLS 1.3. For TLS 1.3, we currently cannot affirmatively answer the question of whether it provides provable strong key independence in the 0-RTT mode or not. This is due to the fact that the key derivation procedure, and thus the protocol specification, is not yet finalized. However, the use of digital signatures for authentication of the server's Diffie-Hellman share at least seems to mitigate a theoretical attack along the lines of the one described above.

Previous work on LLKE. The concept of LLKE was not developed in academia, but in industry – motivated by concrete practical demands of distributed applications. All previous works on LLKE [9,17] conducted *a-posteriori* security analyses of the QUIC protocol. There are no foundational constructions yet, and the relation to other cryptographic protocols and primitives is not yet well-understood.

At ACM CCS 2014, Fischlin and Günther [9] provided a formal definition of *multi-stage* key exchange protocols and used it to analyze the security of QUIC. Lychev *et al.* [17] gave an alternate analysis of QUIC, which considers both efficiency and security. They describe a security model which is tailored specifically to QUIC, adopting the complex, monolithic security model of [11] to the protocol's requirements.

Security model. In this paper, we use a variant of the Canetti-Krawczyk [6] security model. This family of security models is especially suited to protocols with only two message exchanges, with *one-round* key exchange protocols being the most important subclass. Popular examples of such protocols are MQV [16], HMQV [12], SMQV [20], KEA [18,15], and NAXOS [14]. A comparison of different variants of the Canetti-Krawczyk model can be found in [8,22].

The CK model class provides "implicit authentication" (instead of Bellare-Rogaway-style [2] "explicit authentication"), where session keys k_1 and k_2 can only be computed by authenticated parties knowing a (semi-) static key. As an additional security property, we require *strong key independence*, which is captured in our security models.

The importance of simplicity of security models. Security models for key exchange protocols have to consider *active* adversaries that may modify, replay, inject, drop, etc., any message transmitted between communicating parties. They also need to capture *parallel* executions of multiple protocol sessions, potential reveals of earlier session keys, and adaptive corruptions of long-term secrets of parties. This makes even standard security models for key exchange *extremely* complex (in comparison to most other standard cryptographic primitives, like digital signatures or public-key encryption, for example).

Naturally, the novel primitive of LLKE requires formal security definitions. There are different ways to create such a model. One approach is to focus on *generality* of the model. Fischlin and Günther [9] followed this path, by defining *multi-stage* key exchange protocols, a generalization of LLKE. This approach has the advantage that it lays the foundation for the study of a very general class of interesting novel primitives. However, its drawback is that this generality inherently also brings a huge *complexity* to the model. Clearly, the more complex the security model, the more difficult it becomes to devise new, simple, efficient, and provably-secure constructions. Moreover, proofs in complex models tend to be error-prone and less intuitive, because central technical ideas may be concealed in formal details that are required to handle the generality of the model.

Another approach is to devise a model which is tailored to the analysis of *one specific protocol*. For example, the complex, monolithic ACCE security model was developed in [11] to provide an *a posteriori* security analysis of TLS.[6] A similar approach was followed by Lychev *et al.* [17], who adopted this model for an *a posteriori* analysis of QUIC, by defining the so-called Q-ACCE model. A drawback of this approach is that such tailor-made models tend to capture only the properties achieved by existing protocols, but not necessarily all properties that we would expect from a "good" LLKE protocol. In general, such tailor-made models do not, therefore, form a useful foundation for the creation of new protocols.

In this paper, we follow a different approach. We propose novel "bare-bone" security models for LLKE, which aim at capturing *all*, but also *only* the properties expected from "good" LLKE protocols. We propose two different models. One considers the practically-relevant case of *server-only* authentication (where the client may or may not authenticate later over the established communication channel, similar in spirit to the server-only-authenticated ACCE model of [13]). The other considers traditional *mutual* cryptographic authentication of a client and server.

The reduced generality of our definitions – in comparison to the very general multi-stage security model of [9] – is intended. A model which captures *only*, but also *all* the properties expected from a "good" LLKE protocol allows us to devise relatively simple, foundational, and generic constructions of LLKE protocols with as-clean-as-possible security analyses.

Importance of foundational generic constructions. Following [3], we use non-interactive key exchange (NIKE) [7,10] in combination with digital signatures as a main building block.[7] This yields the first examples of LLKE protocols with strong key independence, as well as the first constructions of LLKE from generic complexity assumptions. There are many advantages of such generic constructions:

1. Generic constructions provide a better understanding of the structure of protocols. Since the primitives we use have abstract security properties, we can directly see which abstract security requirements are needed to implement LLKE protocols.
2. They clarify the relations and implications between different types of cryptographic primitives.
3. They can generically be instantiated with building blocks based on different complexity assumptions. For example, if "post-quantum" security is needed, one can directly obtain a concrete protocol by using only post-quantum secure building blocks in the generic construction.

Generally, generic constructions tend to involve more computational overhead than ad-hoc constructions. However, we note that our LLKE protocols can be

---

[6] A more modular approach was later proposed in [4].

[7] Recall that digital signatures are implied by one-way functions, which in turn are implied by NIKE. Thus, essentially we only assume the existence of NIKE as a building block.

instantiated relatively efficiently, given the efficient NIKE schemes of [10], for example.

Contributions. Contributions in this paper can be summarized as follows:

- *Simple security models.* We provide simple security models, which capture all properties that we expect from a "good" LLKE protocol, but only these properties. We consider both the "practical" setting with server-only authentication and the classical setting with mutual authentication.
- *First generic constructions.* We give intuitive, relatively simple, and efficient constructions of LLKE protocols in both settings.
- *Non-DH instantiation.* Both QUIC and TLS 1.3 are based on Diffie-Hellman key exchange. Our generic construction yields the first LLKE protocol which is not based on Diffie-Hellman (e.g., by instantiating the generic construction with the factoring-based NIKE scheme of Freire *et al.* [10]).
- *First LLKE with strong key independence.* Our LLKE protocols are the first to achieve strong key independence in the sense described above.
- *Well-established, general assumptions.* The construction is based on general assumptions, namely the existence of secure NIKE and digital signature schemes. For all building blocks we require only standard security properties.
- *Security in the Standard Model.* The security analysis is completely in the standard model, i.e. it is performed without resorting to the Random Oracle heuristic [1] and without relying on non-standard complexity assumptions.
- *Efficient instantiability.* Despite the fact that our constructions are generic, the resulting protocols can be instantiated relatively efficiently.

2 Preliminaries

For our construction in Section 6, we need signature schemes and non-interactive key exchange (NIKE) protocols. Here we summarize the definitions of these two primitives and their security from the literature.

2.1 Digital Signatures

A digital signature scheme consists of three polynomial-time algorithms $\mathsf{SIG} = (\mathsf{SIG.Gen}, \mathsf{SIG.Sign}, \mathsf{SIG.Vfy})$. The key generation algorithm $(sk, pk) \overset{\$}{\leftarrow} \mathsf{SIG.Gen}(1^\lambda)$ generates a public verification key pk and a secret signing key sk on input of security parameter λ. Signing algorithm $\sigma \overset{\$}{\leftarrow} \mathsf{SIG.Sign}(sk, m)$ generates a signature for message m. Verification algorithm $\mathsf{SIG.Vfy}(pk, \sigma, m)$ returns 1 if σ is a valid signature for m under key pk, and 0 otherwise.

Consider the following security experiment played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.

1. The challenger generates a public/secret key pair $(sk, pk) \overset{\$}{\leftarrow} \mathsf{SIG.Gen}(1^\lambda)$, the adversary receives pk as input.

2. The adversary may query arbitrary messages m_i to the challenger. The challenger replies to each query with a signature $\sigma_i = \mathsf{SIG.Sign}(sk, m_i)$. Here i is an index, ranging between $1 \leq i \leq q$ for some $q \in \mathbb{N}$. Queries can be made adaptively.
3. Eventually, the adversary outputs a message/signature pair (m, σ).

Definition 1. *We define the advantage on an adversary $\mathcal{A}$ in this game as*

$$\mathsf{Adv}^{sEUF\text{-}CMA}_{\mathsf{SIG},\mathcal{A}}(\lambda) := \Pr\left[(m,\sigma) \xleftarrow{\$} \mathcal{A}^{\mathcal{C}(\lambda)}(pk) : \begin{array}{c} \mathsf{SIG.Vfy}(pk,\sigma,m) = 1, \\ (m,\sigma) \neq (m_i,\sigma_i)\ \forall i \end{array}\right] .$$

SIG *is* strongly secure *against* existential forgeries under adaptive chosen-message attacks *(sEUF-CMA), if* $\mathsf{Adv}^{sEUF\text{-}CMA}_{\mathsf{SIG},\mathcal{A}}(\lambda)$ *is a negligible function in* λ *for all probabilistic polynomial-time adversaries* $\mathcal{A}$*.*

Remark 1. Signatures with sEUF-CMA security can be constructed generically from any EUF-CMA-secure signature scheme and chameleon hash functions [5,21].

2.2 Secure Non-Interactive Key Exchange

Definition 2. *A non-interactive key exchange (NIKE) scheme consists of two deterministic algorithms* $(\mathsf{NIKEgen}, \mathsf{NIKEkey})$*.*

$\mathsf{NIKEgen}(1^\lambda, r)$ *takes a security parameter* λ *and randomness* $r \in \{0,1\}^\lambda$*. It outputs a key pair* (pk, sk)*. We write* $(pk, sk) \xleftarrow{\$} \mathsf{NIKEgen}(1^\lambda)$ *to denote that* $\mathsf{NIKEgen}(1^\lambda, r)$ *is executed with uniformly random* $r \xleftarrow{\$} \{0,1\}^\lambda$*.*

$\mathsf{NIKEkey}(sk_i, pk_j)$ *is a* deterministic *algorithm which takes as input a secret key* sk_i *and a public key* pk_j*, and outputs a key* $k_{i,j}$*.*

We say that a NIKE scheme is correct*, if for all* $(pk_i, sk_i) \xleftarrow{\$} \mathsf{NIKEgen}(1^\lambda)$ *and* $(pk_j, sk_j) \xleftarrow{\$} \mathsf{NIKEgen}(1^\lambda)$ *holds that* $\mathsf{NIKEkey}(sk_i, pk_j) = \mathsf{NIKEkey}(sk_j, pk_i)$*.*

A NIKE scheme is used by d parties $P_1, \ldots, P_d$ as follows. Each party P_i generates a key pair $(pk_i, sk_i) \leftarrow \mathsf{NIKEgen}(1^\lambda)$ and publishes pk_i. In order to compute the key shared by P_i and P_j, party P_i computes $k_{i,j} = \mathsf{NIKEkey}(sk_i, pk_j)$. Similarly, party P_j computes $k_{j,i} = \mathsf{NIKEkey}(sk_j, pk_i)$. Correctness of the NIKE scheme guarantees that $k_{i,j} = k_{j,i}$.

CKS-light security. The *CKS-light* security model for NIKE protocols is relatively simplistic and compact. We choose this model because other (more complex) NIKE security models like *CKS*, *CKS-heavy* and *m-CKS-heavy* are polynomial-time equivalent to *CKS-light*. See [10] for more details.

Security of a NIKE protocol NIKE is defined by a game **NIKE** played between an adversary $\mathcal{A}$ and a challenger. The challenger takes a security parameter λ and a random bit b as input and answers all queries of $\mathcal{A}$ until she outputs a bit b'. The challenger answers the following queries for $\mathcal{A}$:

- $\mathsf{RegisterHonest}(i)$. $\mathcal{A}$ supplies an index i. The challenger runs $\mathsf{NIKEgen}(1^\lambda)$ to generate a key pair (pk_i, sk_i) and records the tuple $(\texttt{honest}, pk_i, sk_i)$ for later and returns pk_i to $\mathcal{A}$. This query may be asked *at most twice* by $\mathcal{A}$.
- $\mathsf{RegisterCorrupt}(pk_i)$. With this query $\mathcal{A}$ supplies a public key pk_i. The challenger records the tuple $(\texttt{Corrupt}, pk_i)$ for later.
- $\mathsf{GetCorruptKey}(i, j)$. $\mathcal{A}$ supplies two indexes i and j where pk_i was registered as corrupt and pk_j as honest. The challenger runs $k \leftarrow \mathsf{NIKEkey}(sk_j, pk_i)$ and returns k to $\mathcal{A}$.
- $\mathsf{Test}(i, j)$. The adversary supplies two indexes i and j that were registered honestly. Now the challenger uses bit b: if $b = 0$, then the challenger runs $k_{i,j} \leftarrow \mathsf{NIKEkey}(pk_i, sk_j)$ and returns the key $k_{i,j}$. If $b = 1$, then the challenger samples a random element from the key space, records it for later, and returns the key to $\mathcal{A}$.

The game **NIKE** outputs 1, denoted by $\mathbf{NIKE}^{\mathcal{A}}_{\mathsf{NIKE}}(\lambda) = 1$, if $b = b'$ and 0 otherwise. We say $\mathcal{A}$ wins the game if $\mathbf{NIKE}^{\mathcal{A}}_{\mathsf{NIKE}}(\lambda) = 1$.

Definition 3. *For any adversary $\mathcal{A}$ playing the above* **NIKE** *game against a NIKE scheme* NIKE*, we define the advantage of winning the game* **NIKE** *as*

$$\mathtt{Adv}^{\textit{CKS-light}}_{\mathsf{NIKE},\mathcal{A}}(\lambda) = \left| \Pr\left[\mathbf{NIKE}^{\mathcal{A}}_{\mathsf{NIKE}}(\lambda) = 1\right] - \frac{1}{2} \right| .$$

Let λ be a security parameter, NIKE *be a NIKE protocol and $\mathcal{A}$ an adversary. We say* NIKE *is a CKS-light-*secure *NIKE protocol, if for all probabilistic polynomial-time adversaries $\mathcal{A}$, the function $\mathtt{Adv}^{\textit{CKS-light}}_{\mathsf{NIKE},\mathcal{A}}(\lambda)$ is a negligible function in λ.*

3 Low-Latency Key Exchange Protocols: Syntax and Security with Server-only Authentication

In the model presented in this section, we give formal definitions for LLKE with forward secrecy and strong key independence. We start with the case of server-only authentication, as it is the more important case in practice (in particular, server-only authentication will be the main operating mode of both QUIC and TLS 1.3).

3.1 Syntax and Correctness

Definition 4. *A low-latency key exchange (LLKE) scheme with server-only authentication consists of deterministic algorithms* $(\mathsf{Gen}^{\mathsf{server}}, \mathsf{KE}^{\mathsf{client}}_{\mathsf{init}}, \mathsf{KE}^{\mathsf{client}}_{\mathsf{refresh}}, \mathsf{KE}^{\mathsf{server}}_{\mathsf{refresh}})$.

- $\mathsf{Gen}^{\mathsf{server}}(1^\lambda, r) \rightarrow (pk, sk)$*: A key generation algorithms that takes as input a security parameter λ and randomness $r \in \{0,1\}^\lambda$ and outputs a key pair (pk, sk). We write $(pk, sk) \overset{\$}{\leftarrow} \mathsf{Gen}^{\mathsf{server}}(1^\lambda)$ to denote that a pair (pk, sk) is the output of $\mathsf{Gen}^{\mathsf{server}}$ when executed with uniformly random $r \overset{\$}{\leftarrow} \{0,1\}^\lambda$.*

- $\mathsf{KE}^{\mathsf{client}}_{\mathsf{init}}(pk_j, r_i) \to (k^{i,j}_{\mathtt{tmp}}, m_i)$: *An algorithm that takes as input a public key* pk_j *and randomness* $r_i \in \{0,1\}^\lambda$, *and outputs a temporary key* $k^{i,j}_{\mathtt{tmp}}$ *and a message* m_i.
- $\mathsf{KE}^{\mathsf{server}}_{\mathsf{refresh}}(sk_j, r_j, m_i) \to (k^{j,i}_{\mathtt{main}}, k^{j,i}_{\mathtt{tmp}}, m_j)$: *An algorithm that takes as input a secret key* sk_j, *randomness* r_j *and a message* m_i, *and outputs a key* $k^{j,i}_{\mathtt{main}}$, *a temporary key* $k^{j,i}_{\mathtt{tmp}}$ *and a message* m_j.
- $\mathsf{KE}^{\mathsf{client}}_{\mathsf{refresh}}(pk_j, r_i, m_j) \to k^{i,j}_{\mathtt{main}}$: *An algorithm that takes as input a public key* pk_j, *randomness* r_i, *and message* m_j, *and outputs a key* $k^{i,j}_{\mathtt{main}}$.

We say that a low-latency key exchange scheme is correct, *if for all* (pk_j, sk_j), $\xleftarrow{\$} \mathsf{Gen}^{\mathsf{server}}(1^\lambda)$ *and for all* $r_i, r_j \xleftarrow{\$} \{0,1\}^\lambda$ *holds that*

$$\Pr[k^{i,j}_{\mathtt{tmp}} \neq k^{j,i}_{\mathtt{tmp}} \text{ or } k^{i,j}_{\mathtt{main}} \neq k^{j,i}_{\mathtt{main}}] \leq \mathsf{negl}(\lambda)\ ,$$

where $(k^{j,i}_{\mathtt{tmp}}, m_i) \leftarrow \mathsf{KE}^{\mathsf{client}}_{\mathsf{init}}(pk_j, r_i)$, $(k^{i,j}_{\mathtt{tmp}}, k^{i,j}_{\mathtt{main}}, m_j) \leftarrow \mathsf{KE}^{\mathsf{server}}_{\mathsf{refresh}}(sk_j, r_j, m_i)$, *and* $k^{j,i}_{\mathtt{main}} \leftarrow \mathsf{KE}^{\mathsf{client}}_{\mathsf{refresh}}(pk_j, r_i, m_j)$.

A LLKE scheme is used by a set parties which are either clients C or servers S (cf. Figure 2). Each server S_p, has a generated key pair $(sk_p, pk_p) \xleftarrow{\$} \mathsf{Gen}^{\mathsf{server}}(1^\lambda, j)$ with published pk_p. The protocol is executed as follows:

1. The client oracle C_i chooses $r_i \in \{0,1\}^\lambda$ and selects the public key of the intended partner S_j, (which must be a server, otherwise this value is undefined). Then it computes $(k^{i,j}_{\mathtt{tmp}}, m_i) \leftarrow \mathsf{KE}^{\mathsf{client}}_{\mathsf{init}}(pk_j, r_i)$, and sends m_i to S_j. Additionally, C_i can use $k^{i,j}_{\mathtt{tmp}}$ to encrypt some data M_i.
2. Upon reception of message m_i, S_j, initializes a new oracle $\mathsf{S}_{j,t}$. This oracle chooses $r_j \in \{0,1\}^\lambda$ and computes $(k^{j,i}_{\mathtt{main}}, k^{j,i}_{\mathtt{tmp}}, m_j) \leftarrow \mathsf{KE}^{\mathsf{server}}_{\mathsf{refresh}}(sk_j, r_j, m_i)$. The server may use the ephemeral key $k^{j,i}_{\mathtt{tmp}}$ to decrypt D_i. Then, the server sends m_j and optionally some data M_j encrypted with the key $k^{j,i}_{\mathtt{main}}$ to the client.
3. C_i computes $k^{i,j}_{\mathtt{main}} \leftarrow \mathsf{KE}^{\mathsf{client}}_{\mathsf{refresh}}(pk_j, r_i, m_j)$ and can optionally decrypt D_j. Correctness of the LLKE scheme guarantees that $k^{i,j}_{\mathtt{main}} = k^{j,i}_{\mathtt{main}}$.

3.2 Execution Environment

We provide an adversary $\mathcal{A}$ against an LLKE protocol with the following *execution environment.* Clients, which are not in possession of a long-term secret are represented by oracles $\mathsf{C}_1, \ldots, \mathsf{C}_d$ (without any particular "identity"). We consider l servers, each server has a long-term key pair (sk_j, pk_j)[8], $j \in \{1, \ldots, l\}$,

---

[8] We do not distinguish between static (i.e. long-lived) and semi-static (i.e. medium lived) key pairs. Thus the long-lived keys in this model correspond to the server configuration file keys of QUIC and TLS 1.3.

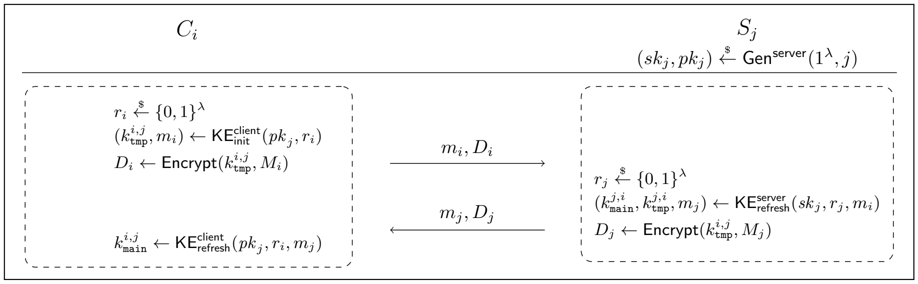

Fig. 2: Execution of a LLKE Protocol with Server-Only Authentication

and each client has access to all public keys $pk_1, \ldots, pk_\ell$. Each server is represented by a collection of k oracles $\mathsf{S}_{j,1}, \ldots, \mathsf{S}_{j,k}$, where each oracle represents a process that executes one single instance of the protocol.

We use the following variables to maintain the internal state of oracles.

Clients. Each client oracle C_i, $i \in [d]$, maintains

- two variables $\mathsf{k}_i^{\mathsf{tmp}}$ and $\mathsf{k}_i^{\mathsf{main}}$ to store the temporal and main keys of a session,
- a variable $\mathsf{Partner}_i$, which contains the identity of the intended communication partner, and
- variables $\mathcal{M}_i^{\mathbf{in}}$ and $\mathcal{M}_i^{\mathbf{out}}$ containing messages sent and received by the oracle.

The internal state of a client oracle is initialized to $(\mathsf{k}_i^{\mathsf{tmp}}, \mathsf{k}_i^{\mathsf{main}}, \mathsf{Partner}_i, \mathcal{M}_i^{\mathbf{in}}, \mathcal{M}_i^{\mathbf{out}}) := (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$.

Servers. Each server oracle $\mathsf{S}_{j,t}$, $(j,t) \in [\ell] \times [k]$, maintains:

- two variables $\mathsf{k}_i^{\mathsf{tmp}}$ and $\mathsf{k}_i^{\mathsf{main}}$ to store the temporal and main keys of a session, and
- variables $\mathcal{M}_{j,t}^{\mathbf{in}}$ and $\mathcal{M}_{j,t}^{\mathbf{out}}$ containing messages sent and received by the server.

The internal state of a server oracle is initialized to $(\mathsf{k}_{j,t}^{\mathsf{tmp}}, \mathsf{k}_{j,t}^{\mathsf{main}}, \mathcal{M}_{j,t}^{\mathbf{in}}, \mathcal{M}_{j,t}^{\mathbf{out}}) := (\emptyset, \emptyset, \emptyset, \emptyset)$.

We say that an oracle has *accepted the temporal key* if $\mathsf{k}^{\mathsf{tmp}} \neq \emptyset$, and *accepted the main key* if $\mathsf{k}^{\mathsf{main}} = \emptyset$.

In the security experiment, the adversary is able to interact with the oracles by issuing the following queries.

$\mathtt{Send}(\mathsf{C}_i/\mathsf{S}_{j,t}, m)$**.** The adversary sends a message m to the requested oracle. The oracle processes m according to the protocol specification. Any response generated by the oracle according to the protocol specification is returned to the adversary.

If a client oracle C_i receives m as the first message, then the oracle checks if m consists of a special initialization message $(m = (\mathtt{init}, j))$. If true, then the oracle responds with the first protocol message generated for intended partner $\mathsf{S}_{j,}$, else it outputs $\perp$.

`Reveal`($\mathsf{C}_i/\mathsf{S}_{j,t}$, `tmp/main`). This query returns the key of the given stage if it already has been computed, or $\perp$ otherwise.

`Corrupt`(j). On input of a server identity j, this query returns the long-term private key of the server. If `Corrupt`(j) is the τ-th query issued by $\mathcal{A}$, we say a party is *τ-corrupted.* For parties that are not corrupted we define $\tau := \infty$.

`Test`($\mathsf{C}_i/\mathsf{S}_{j,t}$, `tmp/main`). This query is used to test a key and is only asked once. It is answered as follows: If the variable of the requested key is not empty, a random $b \xleftarrow{\$} \{0,1\}$ is selected, and
- if $b = 0$ then the requested key is returned, else
- if $b = 1$ then a random key, according to the probability distribution of keys generated by the protocol, is returned.

Otherwise $\perp$ is returned.

3.3 Security Model

Security Game $\mathcal{G}_{\mathcal{A}}^{\mathcal{LLKE}-sa}$. After receiving a security parameter λ the challenger $\mathcal{C}$ simulates the protocol and keeps track of all variables of the execution environment: he generates the long-lived key pairs of all server parties, and answers faithfully to all queries by the adversary.

The adversary receives all public keys $pk_1, \ldots, pk_l$ and can interact with the challenger by issuing any combination of the queries `Send`(), `Corrupt`(), and `Reveal`(). At some point the adversary queries `Test`() to an oracle and receives a key, which is either the requested key or a random value. The adversary may continue asking `Send`(), `Corrupt`(), and `Reveal`()-queries after receiving the bit and finally outputs some bit b'.

Definition 5 (LLKE-Security with Server-Only Authentication). *Let an adversary $\mathcal{A}$ interact with the challenger in game $\mathcal{G}_{\mathcal{A}}^{\mathcal{LLKE}-sa}$ as it is described above. We say the challenger outputs 1, denoted by $\mathcal{G}_{\mathcal{A}}^{\mathcal{LLKE}-sa}(\lambda) = 1$, if $b = b'$ and the following conditions hold:*

- *if $\mathcal{A}$ issues* `Test`(C_i, `tmp`) *all of the following hold:*
  - `Reveal`(C_i, `tmp`) *was never queried by $\mathcal{A}$*
  - `Reveal`($\mathsf{S}_{j,t}$, `tmp`) *was never queried by $\mathcal{A}$ for any oracle $\mathsf{S}_{j,t}$ such that* $\mathsf{Partner}_i = j$ *and* $\mathcal{M}_{j,t}^{\boldsymbol{in}} = \mathcal{M}_i^{\boldsymbol{out}}$
  - *the communication partner* $\mathsf{Partner}_i = j$*, if it exists, is not* τ-corrupted *with* $\tau < \infty$
- *if $\mathcal{A}$ issues* `Test`(C_i, `main`) *all of the following hold:*
  - `Reveal`(C_i, `main`) *was never queried by $\mathcal{A}$*
  - `Reveal`($\mathsf{S}_{j,t}$, `main`) *was never queried by $\mathcal{A}$, where* $\mathsf{Partner}_i = j$, $\mathcal{M}_{j,t}^{\boldsymbol{in}} = \mathcal{M}_i^{\boldsymbol{out}}$, *and* $\mathcal{M}_i^{\boldsymbol{in}} = \mathcal{M}_{j,t}^{\boldsymbol{out}}$
  - *the communication* $\mathsf{Partner}_i = j$ *is not* τ-corrupted *with* $\tau < \tau_0$*, where* `Test`(C_i, `main`) *is the τ_0-th query issued by $\mathcal{A}$*
- *if $\mathcal{A}$ issues* `Test`($\mathsf{S}_{j,t}$, `tmp`) *all of the following hold:*

  - $\texttt{Reveal}(\mathsf{S}_{j,t}, \texttt{tmp})$ *was never queried by* $\mathcal{A}$
  - *there exists an oracle* C_i *with* $\mathcal{M}_i^{\boldsymbol{out}} = \mathcal{M}_{j,t}^{\boldsymbol{in}}$
  - $\texttt{Reveal}(\mathsf{C}_i, \texttt{tmp})$ *was never queried by* $\mathcal{A}$ *to any oracle* C_i *with* $\mathcal{M}_i^{\boldsymbol{out}} = \mathcal{M}_{j,t}^{\boldsymbol{in}}$
  - $\texttt{Reveal}(\mathsf{S}_{j,t'}, \texttt{tmp})$ *was never queried by* $\mathcal{A}$ *for any oracle* $\mathsf{S}_{j,t'}$ *with* $\mathcal{M}_{j,t}^{\boldsymbol{in}} = \mathcal{M}_{j,t'}^{\boldsymbol{in}}$
  - $\mathsf{S}_{j,}$ *is not* τ-corrupted *with* $\tau < \infty$
- *if* $\mathcal{A}$ *issues* $\texttt{Test}(\mathsf{S}_{j,t}, \texttt{main})$ *all of the following hold:*
  - $\texttt{Reveal}(\mathsf{S}_{j,t}, \texttt{main})$ *was never queried by* $\mathcal{A}$
  - *there exists an oracle* C_i *with* $\mathcal{M}_i^{\boldsymbol{out}} = \mathcal{M}_{j,t}^{\boldsymbol{in}}$
  - $\texttt{Reveal}(\mathsf{C}_i, \texttt{main})$ *was never queried by* $\mathcal{A}$*, if* $\mathcal{M}_i^{\boldsymbol{in}} = \mathcal{M}_{j,t}^{\boldsymbol{out}}$

else the game outputs a random bit. We define the advantage of $\mathcal{A}$ *in the game* $\mathcal{G}_{\mathcal{A}}^{\mathcal{LLKE}-sa}(\lambda)$ *by*

$$\texttt{Adv}_{\mathcal{A}}^{\mathcal{LLKE}-sa}(\lambda) := \left| \Pr[\mathcal{G}_{\mathcal{A}}^{\mathcal{LLKE}-sa}(\lambda) = 1] - \frac{1}{2} \right| .$$

Definition 6. *We say that a low-latency key exchange protocol is* `test`*-secure, if there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that for all PPT adversaries* $\mathcal{A}$ *interacting according to the security game* $\mathcal{G}_{\mathcal{A}}^{\mathcal{LLKE}-sa}(\lambda)$ *it holds that*

$$\texttt{Adv}_{\mathcal{A}}^{\mathcal{LLKE}-sa}(\lambda) \leq \mathsf{negl}(\lambda) .$$

Remark 2. Our security model captures forward secrecy, because key indistinguishability is required to hold even if the adversary is able to corrupt the communication partner of the `test`-oracle (but only after the `test`-oracle has accepted, of course, in order to avoid trivial attacks).

Moreover, strong key independence is modeled by the fact that an adversary which attempts to distinguish a `tmp`-key from random (i.e., an adversary which asks $\mathsf{Test}(X, \texttt{tmp})$ for $X \in \{\mathsf{C}_i, \mathsf{S}_{j,t}$ for some $i, j, t\}$) is allowed to learn the `main`-key of X. Similarly, an adversary which tries to distinguish a `main`-key from random by asking $\mathsf{Test}(X, \texttt{main})$ is allowed to learn the `tmp`-key of X as well. Security in this sense guarantees that the `tmp`-key and the `main`-key look independent to a computationally-bounded adversary.

4 Generic Construction of LLKE from NIKE

Now we are ready to describe our generic NIKE-based LLKE protocol and its security analysis.

4.1 Generic Construction

Let $\mathsf{NIKE} = (\mathsf{NIKEgen}, \mathsf{NIKEkey})$ be a NIKE scheme according to Definition 2 and let $\mathsf{SIGN} = (\mathsf{SIG.Gen}, \mathsf{SIG.Sign}, \mathsf{SIG.Vfy})$ be a signature scheme. Then we construct a LLKE scheme $\mathsf{LLKE} = (\mathsf{Gen}^{\mathsf{server}}, \mathsf{KE}^{\mathsf{client}}_{\mathsf{init}}, \mathsf{KE}^{\mathsf{client}}_{\mathsf{refresh}}, \mathsf{KE}^{\mathsf{server}}_{\mathsf{refresh}})$, per Definition 4, in the following way (cf. Figure 3).

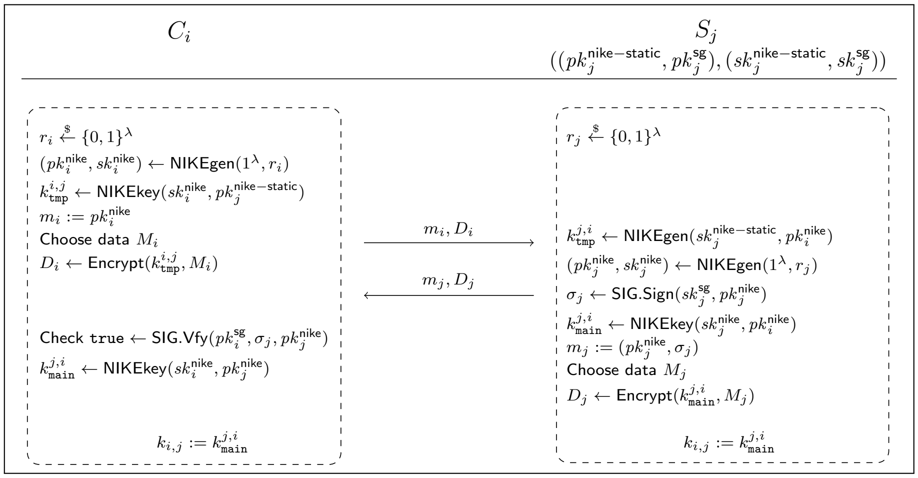

Fig. 3: LLKE from NIKE

- $\mathsf{Gen}^{\mathsf{server}}(1^\lambda, r)$ computes key pairs using the NIKE key generation algorithm $(pk^{\mathsf{nike}}, sk^{\mathsf{nike}}) \xleftarrow{\$} \mathsf{NIKEgen}(1^\lambda)$ and signature keys using the SIGN algorithm $(pk^{\mathsf{sg}}, sk^{\mathsf{sg}}) \xleftarrow{\$} \mathsf{SIG.Gen}$, and outputs
$$(pk, sk) := ((pk^{\mathsf{nike-static}}, pk^{\mathsf{sg}}), (sk^{\mathsf{nike-static}}, sk^{\mathsf{sg}})) \; .$$
- $\mathsf{KE}^{\mathsf{client}}_{\mathsf{init}}(pk_j, r_i)$ samples $r_i \xleftarrow{\$} \{0,1\}^\lambda$, parses $pk_j = (pk_j^{\mathsf{nike-static}}, pk_j^{\mathsf{sg}})$, runs $(pk_i^{\mathsf{nike}}, sk_i^{\mathsf{nike}}) \leftarrow \mathsf{NIKEgen}(1^\lambda, r_i)$ and $k_{i,j}^{\mathsf{nike}} \leftarrow \mathsf{NIKEkey}(sk_i^{\mathsf{nike}}, pk_j^{\mathsf{nike-static}})$, and outputs
$$(k_{\mathtt{tmp}}^{i,j}, m_i) := (k_{i,j}^{\mathsf{nike}}, pk_i^{\mathsf{nike}}) \; .$$
- $\mathsf{KE}^{\mathsf{server}}_{\mathsf{refresh}}(sk_j, r_j, m_i)$ takes in $m_i = pk_i^{\mathsf{nike}}$, parses $sk_j = (sk_j^{\mathsf{nike-static}}, sk_j^{\mathsf{sg}})$, and samples $r_j \xleftarrow{\$} \{0,1\}^\lambda$. It then computes $k_{i,j}^{\mathsf{nike}} \leftarrow \mathsf{NIKEkey}(sk_j^{\mathsf{nike-static}}, pk_i^{\mathsf{nike}})$, $(pk_j^{\mathsf{nike}}, sk_j^{\mathsf{nike}}) \leftarrow \mathsf{NIKEgen}(1^\lambda, r_j)$, and $\sigma_j \leftarrow \mathsf{SIG.Sign}(sk_j^{\mathsf{sg}}, pk_j^{\mathsf{nike}})$. If $m_i = pk_j^{\mathsf{nike-static}}$ then it samples $k_{\mathtt{main}}^{\mathsf{nike}}$ uniformly random, else it computes $k_{\mathtt{main}}^{\mathsf{nike}} \leftarrow \mathsf{NIKEkey}(sk_j^{\mathsf{nike}}, pk_i^{\mathsf{nike}})$, outputting
$$(k_{\mathtt{main}}^{j,i}, k_{\mathtt{tmp}}^{j,i}, m_j) := (k_{\mathtt{main}}^{\mathsf{nike}}, k_{i,j}^{\mathsf{nike}}, (pk_j^{\mathsf{nike}}, \sigma_j)) \; .$$

- $\mathsf{KE}^{\mathsf{client}}_{\mathsf{refresh}}(pk_j, r_i, m_j)$ parses $pk_j = (pk_j^{\mathsf{nike-static}}, pk_j^{\mathsf{sg}})$ and $m_j = (pk_j^{\mathsf{nike}}, \sigma_j)$. It then checks $\texttt{true} \leftarrow \mathsf{SIG.Vfy}(pk_j^{\mathsf{sg}}, \sigma_j, pk_j^{\mathsf{nike}})$ and computes $k^{\mathsf{nike}}_{\mathtt{main}} \leftarrow \mathsf{NIKEkey}(sk_i^{\mathsf{nike}}, pk_j^{\mathsf{nike}})$, outputting
$$k^{i,j}_{\mathtt{main}} := k^{\mathsf{nike}}_{\mathtt{main}} \ .$$

Ultimately, the construction follows by applying the NIKE $\mathsf{NIKEgen}$ algorithm and the SIGN $\mathsf{SIG.Gen}$ algorithm to generate a server configuration file which is comprised of the server public key and a server public signature key which a client can then employ for generating the first protocol flow. In order for the LLKE construction to abstract the security guarantees of the underlying NIKE, the appropriate client $(pk_i^{\mathsf{nike}}, sk_i^{\mathsf{nike}})$ must be available for use in the $\mathsf{NIKEkey}$ algorithm. Consequently, the $(pk_i^{\mathsf{nike}}, sk_i^{\mathsf{nike}})$ values are generated locally by the client, with pk_i^{nike} passed to the server as a message. Note that this construction naturally foregos client-side authentication. Figure 3 demonstrates the construction.

Remark 3. One may wonder why we define $\mathsf{KE}^{\mathsf{server}}_{\mathsf{refresh}}(sk_j, r_j, m_i)$ such that it samples a random key when it takes as input a client message m_i which is equal to its own static NIKE key, that is, if $m_i = pk_j^{\mathsf{nike-static}}$. We note that this is necessary for the security the constructed LLKE scheme to be reducible to that of the NIKE scheme, because in some cases we will not be able to simulate the key computed by a server oracle that receives as input a message which is equal to the "static" NIKE public key contained in its LLKE public key. Note that this incurs a negligible correctness error. However, it is straightforward to verify the correctness of the protocol according to Definition 4.

4.2 Security Proof

We prove security of LLKE in the model of Section 3.3 with server-only authentication.

Theorem 1. *From each attacker $\mathcal{A}$, we can construct attackers $\mathcal{B}_{sig}$, according to Definition 1, and $\mathcal{B}_{nike}$, according to Definition 3, such that*

$$\begin{aligned}\mathtt{Adv}^{\mathcal{LLKE}-sa}_{\mathcal{A}}(\lambda) \leq \quad & 2kdl \cdot \left(\mathtt{Adv}^{CKS\text{-}light}_{\mathsf{NIKE},\mathcal{B}_{nike}}(\lambda) + \mathtt{Adv}^{sEUF\text{-}CMA}_{\mathsf{SIG},\mathcal{B}_{sig}}(\lambda)\right) \\ & + dl \cdot \left(k \cdot \mathtt{Adv}^{CKS\text{-}light}_{\mathsf{NIKE},\mathcal{B}_{nike}}(\lambda) + \mathtt{Adv}^{sEUF\text{-}CMA}_{\mathsf{SIG},\mathcal{B}_{sig}}(\lambda)\right) \\ & + dl \cdot \left(\mathtt{Adv}^{CKS\text{-}light}_{\mathsf{NIKE},\mathcal{B}_{nike}}(\lambda) + \mathtt{Adv}^{sEUF\text{-}CMA}_{\mathsf{SIG},\mathcal{B}_{sig}}(\lambda)\right) \\ & + 4 \cdot \mathtt{Adv}^{CKS\text{-}light}_{\mathsf{NIKE},\mathcal{B}_{nike}}(\lambda) \ .\end{aligned}$$

The running time of $\mathcal{B}_{sig}$ and $\mathcal{B}_{nike}$ is approximately equal to the time required to execute the security experiment with $\mathcal{A}$ once.

Proof. We distinguish between four types of attackers:

- adversary $\mathcal{A}_1$ asks `Test()` to a client oracle and the temporary key (*CT-attacker*)
- adversary $\mathcal{A}_2$ asks `Test()` to a client oracle and the main key (*CM-attacker*)
- adversary $\mathcal{A}_3$ asks `Test()` to a server oracle and the temporary key (*ST-attacker*)
- adversary $\mathcal{A}_4$ asks `Test()` to a server oracle and the main key (*SM-attacker*)

From these, Lemmas 1-4 complete the proof of Theorem 1.

CT-attacker We start with the first attacker that asks $\texttt{Test}(\mathsf{C}_i, \texttt{tmp})$.

Lemma 1. *From each* CT-attacker $\mathcal{A}_1$*, we can construct attackers* $\mathcal{B}_{sig}$*, according to Definition 1, and* $\mathcal{B}_{nike}$*, according to Definition 3, such that*

$$\texttt{Adv}_{\mathcal{A}_1}^{\mathcal{LLKE}-sa}(\lambda) \leq dl \cdot \left(\texttt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) + \texttt{Adv}_{\mathsf{SIG},\mathcal{B}_{sig}}^{sEUF\text{-}CMA}(\lambda)\right) + \texttt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) \ .$$

The running time of $\mathcal{B}_{sig}$ *and* $\mathcal{B}_{nike}$ *is approximately equal to the time required to execute the security experiment with* $\mathcal{A}_1$ *once.*

Proof. The proof is a sequence of different games played between the attacker and a challenger according to the security experiment from Definition 5. Henceforth, let $\texttt{Adv}_i := |\Pr[\texttt{Game}\ i = 1] - 1/2|$ denote the advantage of $\mathcal{A}$ in $\texttt{Game}\ i$.

`Game 0`. This is the original security experiment. By definition we have

$$\texttt{Adv}_0 = \texttt{Adv}_{\mathcal{A}_1}^{\mathcal{LLKE}-sa}(\lambda) \ .$$

`Game 1`. Game 1 is identical to Game 0, except that we add an abort condition. We raise event `abort`, abort the game, and output a random bit, if there ever exist two oracles which compute the same NIKE public key (either in messages or in their long-term public keys). We have

$$\texttt{Adv}_1 \geq \texttt{Adv}_0 - \Pr[\texttt{abort}] \ .$$

Note that in the whole experiment at most $(k+1)l+d$ NIKE keys are generated. By a straightforward reduction to the security of the NIKE scheme, we can construct a trivial NIKE adversary $\mathcal{B}_{nike}$, which retrieves a public key pk^{nike} from the NIKE security experiment, and then generates additional $(k+1)l+d-1$ NIKE key pairs $(pk_i^{\mathsf{nike}}, sk_i^{\mathsf{nike}}) \leftarrow \mathsf{NIKEgen}(1^\lambda, r_i)$, exactly like the security experiment in Game 1. If there exist $i \in [k+d+dl-1]$ with $pk_i^{\mathsf{nike}} = pk^{\mathsf{nike}}$, then $\mathcal{B}_{nike}$ can trivially break the security of the NIKE scheme. Thus we have

$$\Pr[\texttt{abort}] \leq \texttt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda)$$

and therefore

$$\texttt{Adv}_1 \geq \texttt{Adv}_0 - \texttt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) \ .$$

$\texttt{Game 2}$. This game is identical to Game 1 with one exception. We guess $i \xleftarrow{\$} [d]$ uniformly random and let the game abort if $\mathcal{A}_1$ does not issue a $\texttt{Test}(\mathsf{C}_{i'}, \texttt{main})$-query with $i' = i$. That means, in this game we guess the "$\texttt{Test}$-oracle".

Note that we are considering the case of CT-attackers, which always ask a $\texttt{Test}$-query against a client-oracle. Therefore the probability of guessing this oracle correctly is $1/d$, which implies

$$\texttt{Adv}_2 = \frac{1}{d} \cdot \texttt{Adv}_1 \ .$$

$\texttt{Game 3}$. Now, we want to guess the partner of the $\texttt{Test}$-oracle. We choose $j \xleftarrow{\$} [l]$ uniformly random, and abort if $\mathsf{Partner}_i \neq j$. We may assume that C_i "accepts" (as otherwise the $\texttt{Test}$-query returns $\perp$ unconditionally and the adversary cannot win), we must have $\mathsf{Partner}_i \in [l]$ and therefore

$$\texttt{Adv}_3 = \frac{1}{l} \cdot \texttt{Adv}_2 \ .$$

$\texttt{Game 4}$. In this game we add another abort condition to make sure that C_i does not receive the static public key of the server as input. We abort and output a random bit if $\mathcal{M}_i^{\mathbf{in}} = (pk^{\mathsf{nike-static}}, \sigma)$ where $\texttt{true} \leftarrow \mathsf{SIG.Vfy}(pk_j^{\mathsf{sg}}, \sigma, pk^{\mathsf{nike-static}})$, but there exists no $t \in [k]$ with $\mathcal{M}_{j,t}^{\mathbf{out}} = \mathcal{M}_i^{\mathbf{in}}$. Here we can use the fact that the message received by C_i is digitally signed.

Clearly, we have

$$\texttt{Adv}_4 \geq \texttt{Adv}_3 - \Pr[\texttt{abort}'] \ .$$

We claim that we can construct a signature adversary $\mathcal{B}_{sig}$ with $\Pr[\texttt{abort}'] \leq \texttt{Adv}_{\mathsf{SIG},\mathcal{B}_{sig}}^{sEUF\text{-}CMA}(\lambda)$.

$\mathcal{B}_{sig}$ proceeds as follows. It receives as input a public key pk^{sg} and sets $pk_j^{\mathsf{sg}} := pk^{\mathsf{sg}}$. In order to compute signatures to simulate the oracles of server j, $\mathcal{B}_{sig}$ uses the signing oracle provided by the sEUF-CMA security experiment. If event $\texttt{abort}'$ occurs, then this means that C_i receives as input a tuple $\mathcal{M}_i^{\mathbf{in}} = (pk^{\mathsf{nike-static}}, \sigma)$ with $\texttt{true} \leftarrow \mathsf{SIG.Vfy}(pk_j^{\mathsf{sg}}, \sigma, pk^{\mathsf{nike-static}})$, but there exists no server oracle which has output this tuple. Thus, $(pk^{\mathsf{nike-static}}, \sigma)$ is a valid sEUF-CMA forgery for pk_j^{sg}. This proves our claim, and therefore we have

$$\texttt{Adv}_4 \geq \texttt{Adv}_3 - \texttt{Adv}_{\mathsf{SIG},\mathcal{B}_{sig}}^{sEUF\text{-}CMA}(\lambda) \ .$$

The final reduction to the security of the NIKE scheme. We claim that we are now able to construct an efficient attacker $\mathcal{B}_{nike}$ which is able to answer all queries correctly of $\mathcal{A}_1$ such that

$$\texttt{Adv}_4 \leq \texttt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) \ .$$

$\mathcal{B}_{nike}$ interacts with the challenger exactly as it is describe in Definition 3 and runs $\mathcal{A}$ as subroutine, by simulating the experiment as it is described in Game 4. In the reduction, $\mathcal{B}_{nike}$ registers two honest parties P_i and P_j and receives the public keys $\{pk_i^{\mathsf{nike}}, pk_j^{\mathsf{nike}}\}$. $\mathcal{B}_{nike}$ sets the public key of $\mathsf{S}_{j,t}$ to $pk_j^{\mathsf{nike-static}} = pk_j^{\mathsf{nike}}$ and generates the signing keys $\mathsf{S}_{j,t}$. Then, $\mathcal{B}_{nike}$ sets the first message of C_i to $m_i = pk_i^{\mathsf{nike}}$. Next, $\mathcal{B}_{nike}$ answers all `Send`()-, `Reveal`()-, and `Corrupt`()-queries of $\mathcal{A}$ as follows.

- `Corrupt`-queries: $\mathcal{A}_1$ asks only `Corrupt`-queries for server oracles $\mathsf{S}_{j',t}$ (see security definition of the model in Definition 5) for $j \neq j'$. $\mathcal{B}_{nike}$ can answer all these queries correctly by using the RegisterCorrupt()-query and the SIG.Gen-algorithm.
- `Reveal`-queries: Here, we have to distinguish between the different keys and stages.
  - `Reveal`($\mathsf{C}_{i'}$, `tmp`): In this case, $\mathcal{B}_{nike}$ is able to reveal all keys for $i' \neq i$, because he can generate the secret keys himself. The query for $i' = i$ is not allowed by the security definition.
  - `Reveal`($\mathsf{C}_{i'}$, `main`): For $i' \neq i$, $\mathcal{B}_{nike}$ can again use the self-generated secret keys. In the case of $i' = i$, Game 1) guarantees that the message received by the client is not equal to the static public key of the server. For all other messages we can use the RegisterCorrupt()-query and the GetCorruptKey()-query.
  - `Reveal`($\mathsf{S}_{j',t}$, `tmp`): If it holds that $\mathcal{M}_{j,t}^{\mathbf{in}} = \mathcal{M}_i^{\mathbf{out}}$ and $j = j'$ then by security definition this query is not allowed. In contrast, the other two cases are addressed as follows. If $j \neq j'$ then $\mathcal{B}_{nike}$ is able to generate all necessary keys to answer the query. For $j = j'$ and $\mathcal{M}_{j,t}^{\mathbf{in}} \neq \mathcal{M}_i^{\mathbf{out}}$ he has to use the RegisterCorrupt()-query and the GetCorruptKey(). $\mathcal{B}_{nike}$ has to generate a random key if $\mathcal{M}_{j,t}^{\mathbf{in}} = pk_j^{\mathsf{nike}}$ to simulate the environment for $\mathcal{A}_1$. This is also defined in the generic construction.
  - `Reveal`($\mathsf{S}_{j',t}$, `main`): In this case, $\mathcal{B}_{nike}$ can generate the secret keys himself to answer the query correctly.
- `Send`-queries: $\mathcal{B}_{nike}$ is able to answer all of this queries using the keys that are self-generated and with the messages answered by the NIKE oracle.

After the `Test`-query $\mathcal{A}_1$ has to get a random value or a key k which depends on the keys $(sk_i^{\mathsf{nike}}, pk_j^{\mathsf{nike-static}})$. This is exactly the same input which $\mathcal{B}_{nike}$ receives after querying Test(i, j) in the NIKE experiment.
Combining all the above games completes the reduction.

CM-attacker The next proof is about attackers that ask `Test`(C_i, `main`).

Lemma 2. *From each* CM-attacker $\mathcal{A}_2$*, we can construct attackers* $\mathcal{B}_{sig}$*, according to Definition 1, and* $\mathcal{B}_{nike}$*, according to Definition 3, such that*

$$\begin{aligned}\mathtt{Adv}_{\mathcal{A}_2}^{\mathcal{LLKE}-sa}(\lambda) \leq \quad & dl \cdot \left(k \cdot \mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{\textit{CKS-light}}(\lambda) + \mathtt{Adv}_{\mathsf{SIG},\mathcal{B}_{sig}}^{\textit{sEUF-CMA}}(\lambda)\right) \\ & + \mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{\textit{CKS-light}}(\lambda) \ .\end{aligned}$$

The running times of $\mathcal{B}_{sig}$ and $\mathcal{B}_{nike}$ are approximately equal to the time required to execute the security experiment with $\mathcal{A}_2$ once.

Proof. Again, we proceed in a sequence of games.

`Game 0`. This is the original security experiment. By definition we have

$$\mathtt{Adv}_0 = \mathtt{Adv}_{\mathcal{A}_2}^{\mathcal{LLKE}-sa}(\lambda)\ .$$

`Game 1`. Game 1 is identical to Game 0, except that we add an abort condition. Like in Game 1 in the proof of Lemma 1, we raise event `abort` and abort the game if there ever exists two oracles which compute the same NIKE key. With exactly the same argument as in Game 1 from the proof of Lemma 1, we have

$$\mathtt{Adv}_0 \leq \mathtt{Adv}_1 + \mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{\mathit{CKS\text{-}light}}(\lambda)\ .$$

`Game 2`. This game is identical to Game 1, except that we guess the "`Test`-oracle". More precisely, we guess an index $i \xleftarrow{\$} [d]$ uniformly at random and abort the game if $\mathcal{A}_2$ does not issue a $\mathtt{Test}(\mathsf{C}_{i'}, \mathtt{main})$-query with $i' = i$.

Note that we are considering the case of CM-attackers, which always ask a `Test`-query against a client-oracle. Therefore the probability of guessing this oracle correctly is $1/d$, which implies

$$\mathtt{Adv}_2 = \frac{1}{d} \cdot \mathtt{Adv}_1\ .$$

`Game 3`. Next, we guess the identity of the partner of oracle C_i. More precisely, we choose $j \xleftarrow{\$} [l]$ uniformly random and abort if $\mathsf{Partner}_i \neq j$. We may assume that C_i "accepts" (as otherwise the `Test`-query returns $\perp$ unconditionally and the adversary cannot win) and thus we must have $\mathsf{Partner}_i \in [l]$. Therefore

$$\mathtt{Adv}_3 = \frac{1}{l} \cdot \mathtt{Adv}_2\ .$$

`Game 4`. Now we want to make sure that there *exists* a server-oracle, which has output the message received by client C_i. Here we can use the fact that the message received by C_i is digitally signed, and that the partner of the `Test`-oracle must not be corrupted before C_i "accepts".

Formally, Game 4 is identical to Game 3, with the exception that we add another abort condition. We raise event `abort'`, let the experiment abort, and output a random bit, if $\mathcal{M}_i^{\mathbf{in}} = (pk_j^{\mathsf{nike}}, \sigma_j)$ where $\mathtt{true} \leftarrow \mathsf{SIG.Vfy}(pk_j^{\mathsf{sg}}, \sigma_j, pk_j^{\mathsf{nike}})$, but there does not exist $t \in [k]$ with $\mathcal{M}_{j,t}^{\mathbf{out}} = \mathcal{M}_i^{\mathbf{in}}$. Clearly, we have

$$\mathtt{Adv}_4 \geq \mathtt{Adv}_3 - \Pr[\mathtt{abort'}]\ .$$

We claim that we can construct a signature adversary $\mathcal{B}_{sig}$ with $\Pr[\mathtt{abort}'] \leq \mathtt{Adv}^{sEUF\text{-}CMA}_{\mathsf{SIG},\mathcal{B}_{sig}}(\lambda)$.

$\mathcal{B}_{sig}$ proceeds as follows. It receives as input a public key pk^{sg} and sets $pk_j^{\mathsf{sg}} := pk^{\mathsf{sg}}$. In order to compute signatures to simulate the oracles of server j, $\mathcal{B}_{sig}$ uses the signing oracle provided by the sEUF-CMA security experiment. If event $\mathtt{abort}'$ occurs, then this means that C_i receives as input a tuple $\mathcal{M}_i^{\mathbf{in}} = (pk_j^{\mathsf{nike}}, \sigma_j)$ with $\mathtt{true} \leftarrow \mathsf{SIG.Vfy}(pk_j^{\mathsf{sg}}, \sigma_j, pk_j^{\mathsf{nike}})$, but there exists no server oracle which has output this tuple. Thus, $(pk_j^{\mathsf{nike}}, \sigma_j)$ is a valid sEUF-CMA forgery for pk_j^{sg}. This proves our claim, and therefore we have

$$\mathtt{Adv}_3 \leq \mathtt{Adv}_4 + \mathtt{Adv}^{sEUF\text{-}CMA}_{\mathsf{SIG},\mathcal{B}_{sig}}(\lambda) \ .$$

`Game 5`. In this game, we guess the partner oracle of C_i, which is guaranteed to exist due to Game 4. That is, we choose $t \overset{\$}{\leftarrow} [k]$ uniformly at random and abort the game if $\mathcal{M}_i^{\mathbf{in}} \neq \mathcal{M}_{j,t}^{\mathbf{out}}$.

Due to Game 4 we know that there exists (j, t') with $\mathcal{M}_i^{\mathbf{in}} = \mathcal{M}_{j,t'}^{\mathbf{out}}$. (Moreover, (j, t') is unique, due to Game 1). Thus, we have $\Pr[t = t'] = 1/k$, and thus

$$\mathtt{Adv}_5 = \frac{1}{k} \cdot \mathtt{Adv}_4 \ .$$

The final reduction to the security of the NIKE scheme. Finally, we claim that we can build $\mathcal{B}_{nike}$, which is able to answer all queries of $\mathcal{A}_2$ and it holds that

$$\mathtt{Adv}_5 \leq \mathtt{Adv}^{CKS\text{-}light}_{\mathsf{NIKE},\mathcal{B}_{nike}}(\lambda) \ .$$

First, $\mathcal{B}_{nike}$ registers two honest parties P_i and P_j and receives the public keys $\{pk_i^{\mathsf{nike}}, pk_j^{\mathsf{nike}}\}$. In this case, $\mathcal{B}_{nike}$ sets the message m_j of $\mathsf{S}_{j,t}$ to $m_j = pk_j^{\mathsf{nike}}$ and the message m_i of C_i to $m_i = pk_i^{\mathsf{nike}}$. Then, $\mathcal{B}_{nike}$ generates all long term keys of the server oracles and answers the queries as follows:

- `Corrupt`-queries: $\mathcal{A}_2$ asks only `Corrupt`-queries for server oracles $\mathsf{S}_{j',t}$ for $j \neq j'$. $\mathcal{B}_{nike}$ can answer all these queries correctly by using the $\mathsf{RegisterCorrupt}()$-query and the $\mathsf{SIG.Gen}$-algorithm. After querying the `Test`-query, the attacker is allowed to receive the long-term keys of $\mathsf{S}_{j',t}$ for $j = j'$.
- `Reveal`-queries: Here, we have to distinguish between the different keys and stages.
  - $\mathtt{Reveal}(\mathsf{C}_{i'}, \mathtt{tmp})$: In this case, $\mathcal{B}_{nike}$ is able to reveal all keys, because he knows all the long term keys of the server oracles.
  - $\mathtt{Reveal}(\mathsf{C}_{i'}, \mathtt{main})$: For $i' \neq i$, $\mathcal{B}_{nike}$ can use again the self-generated secret keys. In the case of $i' = i$, it holds that the queried key depends on the keys sk_i^{nike} and pk_j^{nike}, else the game would abort by definition of `Game1`. For the keys sk_i^{nike} and pk_j^{nike} the attacker $\mathcal{A}_2$ is not allowed to ask the `Reveal`-query.

  - `Reveal`$(\mathsf{S}_{j',t}, \texttt{tmp})$: In this case, $\mathcal{B}_{nike}$ can use the self-generated long term keys of the server to answer the query correctly.
  - `Reveal`$(\mathsf{S}_{j',t}, \texttt{main})$: If $j' = j$ and $\mathcal{M}^{\mathbf{in}}_{j',t} = \mathcal{M}^{\mathbf{out}}_{i}$ then this query is not allowed by the security definition. For all other cases, $\mathcal{B}_{nike}$ can use the $\mathsf{RegisterCorrupt}()$-query and the $\mathsf{GetCorruptKey}()$ to answer the query or the self-generated keys.

- `Send`-queries: $\mathcal{B}_{nike}$ is able to answer all of this queries using the keys that are self-generated and with the messages answered by the NIKE oracle.

Summarily, the last part of the proof follows that of Lemma 1.

ST-attacker We now turn to attackers that ask `Test`$(\mathsf{S}_{j,t}, \texttt{tmp})$.

Lemma 3. *From each* ST-attacker $\mathcal{A}_3$*, we can construct attackers* $\mathcal{B}_{sig}$*, according to Definition 1, and* $\mathcal{B}_{nike}$*, according to Definition 3, such that*

$$\texttt{Adv}^{\mathcal{LLKE}-sa}_{\mathcal{A}_3}(\lambda) \leq kdl\cdot\left(\texttt{Adv}^{CKS\text{-}light}_{\mathsf{NIKE},\mathcal{B}_{nike}}(\lambda) + \texttt{Adv}^{sEUF\text{-}CMA}_{\mathsf{SIG},\mathcal{B}_{sig}}(\lambda)\right) + \texttt{Adv}^{CKS\text{-}light}_{\mathsf{NIKE},\mathcal{B}_{nike}}(\lambda) \ .$$

The running times of $\mathcal{B}_{sig}$ *and* $\mathcal{B}_{nike}$ *are approximately equal to the time required to execute the security experiment with* $\mathcal{A}_3$ *once.*

Proof. Again, we proceed in a sequence of games.

`Game 0`. This is the original security experiment. By definition we have

$$\texttt{Adv}_0 = \texttt{Adv}^{\mathcal{LLKE}-sa}_{\mathcal{A}_3}(\lambda) \ .$$

`Game 1`. Game 1 is identical to Game 0, except that we add an abort condition. We raise event `abort` and abort the game, outputting a random bit, if there ever exists two oracles which compute the same NIKE key.

Reducing to the security of the NIKE scheme as in Game 1 of Lemma 1, yields

$$\texttt{Adv}_0 \leq \texttt{Adv}_1 + \texttt{Adv}^{CKS\text{-}light}_{\mathsf{NIKE},\mathcal{B}_{nike}}(\lambda) \ .$$

`Game 2`. This game is identical to Game 1, except that we guess the "`Test`-oracle" $\mathsf{S}_{j,t}$ via uniformly random indices $(j,t) \xleftarrow{\$} [l] \times [k]$, and abort and output a random bit if the guess is wrong. As before, we have

$$\texttt{Adv}_1 = lk \cdot \texttt{Adv}_2 \ .$$

`Game 3`. Note that there must exist an oracle C_i which has output the message received by $\mathsf{S}_{j,t}$ (by the corresponding condition in the security experiment, which rules out trivial attacks). We guess this "partner" oracle C_i, by choosing $i \xleftarrow{\$} [d]$ uniformly at random and aborting the experiment, outputting a random bit, if $\mathcal{M}_{j,t}^{\mathbf{in}} \neq \mathcal{M}_i^{\mathbf{out}}$. We may assume that $\mathsf{S}_{j,t}$ "accepts" (as otherwise the `Test`-query returns $\perp$ unconditionally and the adversary cannot win). Therefore

$$\mathtt{Adv}_2 = d \cdot \mathtt{Adv}_3 \ .$$

Finally, we claim that we can build $\mathcal{B}_{nike}$, which is able to answer all queries of $\mathcal{A}_3$ and it holds that

$$\mathtt{Adv}_3 \leq \mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) \ .$$

`Game 4`. Now we want to make sure that the client oracle C_i receives only a valid message generated by an oracle of server j as input. We can use the fact that party j must not be corrupted to use the security of the signature scheme as an argument.

Game 4 is identical to Game 3, with the exception that we add another abort condition. We raise event $\mathtt{abort}'$, let the experiment abort, and output a random bit, if $\mathcal{M}_i^{\mathbf{in}} = (pk_j^{\mathsf{nike}}, \sigma_j)$ where $\mathtt{true} \leftarrow \mathsf{SIG.Vfy}(pk_j^{\mathsf{sg}}, \sigma_j, pk_j^{\mathsf{nike}})$, but $\mathcal{M}_{j,t}^{\mathbf{out}} \neq \mathcal{M}_i^{\mathbf{in}}$ for any $t \in [k]$. As in Lemma 1, Game 4, we have

$$\mathtt{Adv}_4 \geq \mathtt{Adv}_3 - \Pr[\mathtt{abort}'] \ ,$$

and claim that we can construct a signature adversary $\mathcal{B}_{sig}$ with $\Pr[\mathtt{abort}'] \leq \mathtt{Adv}_{\mathsf{SIG},\mathcal{B}_{sig}}^{sEUF\text{-}CMA}(\lambda)$.

$\mathcal{B}_{sig}$ proceeds as follows. It receives as input a public signature key pk^{sg} and sets $pk_j^{\mathsf{sg}} := pk^{\mathsf{sg}}$. In order to compute signatures to simulate the oracles of server j, $\mathcal{B}_{sig}$ uses the signing oracle provided by the sEUF-CMA security experiment. If event $\mathtt{abort}'$ occurs, then this means that C_i receives as input a tuple $\mathcal{M}_i^{\mathbf{in}} = (pk_j^{\mathsf{nike}}, \sigma_j)$ with $\mathtt{true} \leftarrow \mathsf{SIG.Vfy}(pk_j^{\mathsf{sg}}, \sigma_j, pk_j^{\mathsf{nike}})$, but which $\mathsf{S}_{j,}$ has not output. Thus, $(pk_j^{\mathsf{nike}}, \sigma_j)$ is a valid sEUF-CMA forgery for pk_j^{sg}. Ergo we have

$$\mathtt{Adv}_3 \leq \mathtt{Adv}_4 + \mathtt{Adv}_{\mathsf{SIG},\mathcal{B}_{sig}}^{sEUF\text{-}CMA}(\lambda) \ .$$

$\mathcal{B}_{nike}$ interacts with $\mathcal{A}_3$ the same way as it interacts in the proof of Lemma 1 with one exception. The query $\mathtt{Reveal}(\mathsf{S}_{j',t'}, \mathtt{tmp})$ with $\mathcal{M}_{j',t'}^{\mathbf{in}} = \mathcal{M}_{j,t}^{\mathbf{in}}$ and $j = j'$ is not allowed (see Definition 5).

Lemma 4. *From each* SM-attacker $\mathcal{A}_4$, *we can construct attackers* $\mathcal{B}_{sig}$, *according to Definition 1, and* $\mathcal{B}_{nike}$, *according to Definition 3, such that*

$$\mathtt{Adv}_{\mathcal{A}_4}^{\mathcal{LLKE}-sa}(\lambda) \leq kdl \cdot \left(\mathtt{Adv}_{\mathsf{SIG},\mathcal{B}_{sig}}^{sEUF\text{-}CMA}(\lambda) + \mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda)\right) + \mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) \ .$$

The running time of $\mathcal{B}_{sig}$ *and* $\mathcal{B}_{nike}$ *is approximately equal to the time required to execute the security experiment with* $\mathcal{A}_4$ *once.*

Proof. As in the proofs of Lemmas 1–3, we use a sequence of games.

`Game 0`. This is the original security experiment. By definition we have

$$\mathtt{Adv}_0 = \mathtt{Adv}_{\mathcal{A}_4}^{\mathcal{LLKE}-sa}(\lambda) \ .$$

`Game 1`. Game 1 is identical to Game 0, except that we add an abort condition. We raise event `abort` and abort the game, outputting a random bit, if there ever exists two oracles which compute the same NIKE key.

Reducing to the security of the NIKE scheme as in Lemma 1, Game 1, yields

$$\mathtt{Adv}_0 \leq \mathtt{Adv}_1 + \mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) \ .$$

`Game 2`. This game is identical to Lemma 2, Game 2 for a $\mathsf{S}_{j,t}$, guessing the "`Test`-oracle" via an index $(j,t) \xleftarrow{\$} [l] \times [k]$ uniformly at random, which gives

$$\mathtt{Adv}_1 = lk \cdot \mathtt{Adv}_2 \ .$$

`Game 3`. Next, we guess the identity of the partner of oracle C_i. More precisely, we choose $i \xleftarrow{\$} [d]$ uniformly at random and abort the experiment, outputting a random bit, if $\mathcal{M}_{j,t}^{\mathbf{in}} \neq \mathcal{M}_i^{\mathbf{out}}$. We may assume that $\mathsf{S}_{j,t}$ "accepts" (as otherwise the `Test`-query returns $\perp$ unconditionally and the adversary cannot win). Therefore

$$\mathtt{Adv}_2 = d \cdot \mathtt{Adv}_3 \ .$$

`Game 4`. Now we want to make sure that the client oracle C_i receives only a valid message which has been output by the server if the message is the ephemeral public key of the server. Game 4 is identical to Game 3, with the exception that we add another abort condition. We raise event `abort'`, let the experiment abort, and output a random bit, if $\mathcal{M}_i^{\mathbf{in}} = (pk_{j,t}^{nike}, \sigma_j)$ where $\mathtt{true} \leftarrow$

$\mathsf{SIG.Vfy}(pk_j^{\mathsf{sg}}, \sigma_j, pk_{j,t}^{\mathsf{nike}})$, but there does not exist $t \in [k]$ with $\mathcal{M}_{j,t}^{\mathbf{out}} = \mathcal{M}_i^{\mathbf{in}}$. Clearly, we have

$$\mathtt{Adv}_4 \geq \mathtt{Adv}_3 - \Pr[\mathtt{abort}'] \ .$$

We claim that we can construct a signature adversary $\mathcal{B}_{sig}$ with $\Pr[\mathtt{abort}'] \leq \mathtt{Adv}_{\mathsf{SIG},\mathcal{B}_{sig}}^{sEUF\text{-}CMA}(\lambda)$.

$\mathcal{B}_{sig}$ proceeds as follows. It receives as input a public key pk^{sg} and sets $pk_j^{\mathsf{sg}} := pk^{\mathsf{sg}}$. In order to compute signatures to simulate the oracles of server j, $\mathcal{B}_{sig}$ uses the signing oracle provided by the sEUF-CMA security experiment. If event $\mathtt{abort}'$ occurs, then this means that C_i receives as input a tuple $\mathcal{M}_i^{\mathbf{in}} = (pk_{j,t}^{\mathsf{nike}}, \sigma_j)$ with $\mathtt{true} \leftarrow \mathsf{SIG.Vfy}(pk_j^{\mathsf{sg}}, \sigma_j, pk_{j,t}^{\mathsf{nike}})$, but there exists no server oracle which has output this tuple. Thus, $(pk_{j,t}^{\mathsf{nike}}, \sigma_j)$ is a valid sEUF-CMA forgery for pk_j^{sg}. This proves our claim, and therefore we have

$$\mathtt{Adv}_3 \leq \mathtt{Adv}_4 + \mathtt{Adv}_{\mathsf{SIG},\mathcal{B}_{sig}}^{sEUF\text{-}CMA}(\lambda) \ .$$

Finally, we claim that we can build $\mathcal{B}_{nike}$, which is able to answer all queries of $\mathcal{A}_4$ and it holds that

$$\mathtt{Adv}_4 \leq \mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) \ .$$

$\mathcal{B}_{nike}$ interacts with $\mathcal{A}_4$ the same way as it interacts in the proof of Lemma 2, except that we allow the corruption of all server oracles.

5 Low-Latency Key Exchange Protocols: Syntax and Security with Mutual Authentication

Building on the work of Section 3, we define LLKE in the context where mutual authentication is possible. Since such a situation requires the presence of client long-term keys, it requires intrinsic assumptions that are not necessary for a general low-latency protocol. Consequently, we separately define protocols where mutual authentication is possible, and provide a corresponding security model that takes this into account.

Definition 7. *A low-latency key exchange with an option for mutual authentication (LLKE-M) scheme consists of four deterministic algorithms* $(\mathsf{Gen}, \mathsf{KE}_{\mathsf{init}}^{\mathsf{client}}, \mathsf{KE}_{\mathsf{refresh}}^{\mathsf{client}}, \mathsf{KE}_{\mathsf{refresh}}^{\mathsf{server}})$.

- $\mathsf{Gen}(1^\lambda, r) \rightarrow (pk, sk)$*: A key generation algorithms that takes as input a security parameter* λ *and randomness* $r \in \{0,1\}^\lambda$ *and outputs a key pair* (pk, sk)*.*

  We write $(pk, sk) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$ *to denote that a pair* (pk, sk) *is the output of* Gen *when executed with uniformly random* $r \xleftarrow{\$} \{0,1\}^\lambda$*.*
- $\mathsf{KE}_{\mathsf{init}}^{\mathsf{client}}(pk_j, sk_i, r_i) \rightarrow (k_{\mathtt{tmp}}^{i,j}, m_i)$*: An algorithm that takes as input a public key* pk_j*, a secret key* sk_i*, and randomness* $r_i \in \{0,1\}^\lambda$*, and outputs a temporary key* $k_{\mathtt{tmp}}^{i,j}$ *and a message* m_i*.*

- $\mathsf{KE}^{\mathsf{server}}_{\mathsf{refresh}}(sk_j, r_j, pk_i, m_i) \to (k^{j,i}_{\mathtt{main}}, k^{j,i}_{\mathtt{tmp}}, m_j)$*: An algorithm that takes as input a secret key* sk_j*, randomness* r_j*, a public key* pk_i*, and a message* m_i*, and outputs a key* $k^{j,i}_{\mathtt{main}}$*, a temporary key* $k^{j,i}_{\mathtt{tmp}}$ *and a message* m_j*.*
- $\mathsf{KE}^{\mathsf{client}}_{\mathsf{refresh}}(pk_j, sk_i, r_i, m_j) \to k^{i,j}_{\mathtt{main}}$*: An algorithm that takes as input a public key* pk_j*, a secret key* sk_i*, randomness* r_i*, and message* m_j*, and outputs a key* $k^{i,j}_{\mathtt{main}}$*.*

We say that a low-latency key exchange scheme is correct*, if for all* (pk_i, sk_i)*,* $(pk_j, sk_j) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$ *and for all* $r_i, r_j \xleftarrow{\$} \{0,1\}^\lambda$ *holds that*

$$\Pr[k^{i,j}_{\mathtt{tmp}} \neq k^{j,i}_{\mathtt{tmp}} \text{ or } k^{i,j}_{\mathtt{main}} \neq k^{j,i}_{\mathtt{main}}] \leq \mathsf{negl}(\lambda) \ ,$$

where $(k^{j,i}_{\mathtt{tmp}}, m_i) \leftarrow \mathsf{KE}^{\mathsf{client}}_{\mathsf{init}}(pk_j, sk_i, r_i)$*,* $(k^{i,j}_{\mathtt{tmp}}, k^{i,j}_{\mathtt{main}}, m_j) \leftarrow \mathsf{KE}^{\mathsf{server}}_{\mathsf{refresh}}(sk_j, r_j, pk_i, m_i)$*, and* $k^{j,i}_{\mathtt{main}} \leftarrow \mathsf{KE}^{\mathsf{client}}_{\mathsf{refresh}}(pk_j, sk_i, r_i, m_j)$*.*

A LLKE-M scheme is used by a set parties which are either clients C or servers S. Each principal has a generated a key pair $(pk_p, sk_p) \xleftarrow{\$} \mathsf{Gen}(1^\lambda, p)$ and with published pk_p. The protocol is executed as follows:

1. The client oracle $\mathsf{C}_{i,s}$ chooses $r_i \in \{0,1\}^\lambda$ and selects the public key of the intended partner $\mathsf{S}_{j,\cdot}$. Then it computes $(k^{i,j}_{\mathtt{tmp}}, m_i) \leftarrow \mathsf{KE}^{\mathsf{client}}_{\mathsf{init}}(pk_j, sk_i, r_i)$, and sends m_i to S_j. Additionally, $\mathsf{C}_{i,s}$ can use $k^{i,j}_{\mathtt{tmp}}$ to encrypt some data M_i.
2. Upon reception of message m_i, $\mathsf{S}_{j,}$ initializes a new oracle $\mathsf{S}_{j,t}$. $\mathsf{S}_{j,t}$ chooses $r_j \in \{0,1\}^\lambda$ and computes $(k^{j,i}_{\mathtt{main}}, k^{j,i}_{\mathtt{tmp}}, m_j) \leftarrow \mathsf{KE}^{\mathsf{server}}_{\mathsf{refresh}}(sk_j, r_j, pk_i, m_i)$. The server may use the ephemeral key $k^{j,i}_{\mathtt{tmp}}$ to decrypt M_i. Then, the server sends m_j and optionally some data M_j encrypted with the key $k^{j,i}_{\mathtt{main}}$ to the client.
3. C_i computes $k^{i,j}_{\mathtt{main}} \leftarrow \mathsf{KE}^{\mathsf{client}}_{\mathsf{refresh}}(pk_j, sk_i, r_i, m_j)$ and can decrypt M_j. Correctness of the LLKE scheme guarantees that $k^{i,j}_{\mathtt{main}} = k^{j,i}_{\mathtt{main}}$.

5.1 Security under Mutual Authentication

In a similar manner to security experiment and model under server-only authentication, we define the experiment and execution environment for the Key-Security game under mutual authentication.

Execution Environment. The security experiment provides the adversary with an execution environment that simulates d clients and ℓ servers. Each client is represented by a collection of n oracles $\mathsf{C}_{i,1}, \ldots, \mathsf{C}_{i,n}$ and every server is represented by a collection of k oracles $\mathsf{S}_{j,1}, \ldots, \mathsf{S}_{j,k}$. Each oracle represents a process that executes one single instance of the protocol. Each principal has a long-term key pair (sk_i, pk_i). We use the following variables to maintain the internal state of oracles. Temporary and main session stages are referenced as `tmp` and `main`.

Each oracle $\mathsf{C}_{i,s}$, $(i,s) \in [d] \times [n]$ (or $\mathsf{S}_{j,t}$, $(j,t) \in [\ell] \times [k]$, respectively), maintains:

- two variables $\mathsf{k}_i^{\mathsf{tmp}}$ and $\mathsf{k}_i^{\mathsf{main}}$ to store the temporal and main keys of a session,
- a variable $\mathsf{Partner}_i$, which contains the identity of the intended communication partner, and
- variables $\mathcal{M}_{i,s}^{\mathbf{in}}$ and $\mathcal{M}_{i,s}^{\mathbf{out}}$ containing messages sent and received by the oracle.

The internal state of an oracle is initialized to $(\mathsf{k}_i^{\mathsf{tmp}}, \mathsf{k}_i^{\mathsf{main}}, \mathsf{Partner}_i, \mathcal{M}_i^{\mathbf{in}}, \mathcal{M}_i^{\mathbf{out}}) := (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$.

We say that an oracle has *accepted the temporal key* if $\mathsf{k}^{\mathsf{tmp}} \neq \emptyset$, and *accepted the main key* if $\mathsf{k}^{\mathsf{main}} = \emptyset$.

5.2 Adversarial Model

Security experiment. Queries allowed to an adversary under the security experiment when mutual authentication is possible correlate to those found in Section 3, with the inputs that an adversary may call the query on modified to the following.

- $\texttt{Send}(\mathsf{C}_{i,s}/\mathsf{S}_{j,t}, m)$
- $\texttt{Reveal}(\mathsf{C}_{i,s}/\mathsf{S}_{j,t}, \texttt{tmp}/\texttt{main})$
- $\texttt{Corrupt}(i/j)$
- $\texttt{Test}(\mathsf{C}_{i,s}/\mathsf{S}_{j,t}, \texttt{tmp}/\texttt{main})$

5.3 Security Model

As in Section 3.3, a challenger follows the key-secrecy game $\mathcal{G}_{\mathcal{A}}^{\mathcal{LLKE}-sa}$, eventually outputting a bit guess b'. However, the win conditions allowed to an adversary are modified as follows, and the game played according to these conditions is denoted $\mathcal{G}_{\mathcal{A}}^{\mathcal{LLKE}-ma}$.

Definition 8 (Key-Secrecy (under Mutual Authentication)). *Let an attacker $\mathcal{A}$ play the game $\mathcal{G}_{\mathcal{A}}^{\mathcal{LLKE}-ma}$ as it is described above. We say the challenger outputs 1, denoted by $\mathcal{G}_{\mathcal{A}}^{\mathcal{LLKE}-sa}(\lambda) = 1$, if $b = b'$ and the following conditions hold:*

- *if $\mathcal{A}$ issues $\texttt{Test}(\mathsf{C}_i, \texttt{tmp})$ all of the following hold:*
  - $\texttt{Reveal}(\mathsf{C}_{i,s}, \texttt{tmp})$ *was never queried by $\mathcal{A}$*
  - $\texttt{Reveal}(\mathsf{S}_{j,t}, \texttt{tmp})$ *was never queried by $\mathcal{A}$, for any oracle $\mathsf{S}_{j,t}$ such that $\mathsf{Partner}_i^s = j$ and $\mathcal{M}_{j,t}^{\boldsymbol{in}} = \mathcal{M}_{i,s}^{\boldsymbol{out}}$*
  - C_i *is not τ-*corrupted *with $\tau < \infty$*
  - *the communication partner $\mathsf{Partner}_i^s = j$, if it exists, is not τ-*corrupted *with $\tau < \infty$*
- *if $\mathcal{A}$ issues $\texttt{Test}(\mathsf{C}_{i,s}, \texttt{main})$ all of the following hold:*
  - $\texttt{Reveal}(\mathsf{C}_{i,s}, \texttt{main})$ *was never queried by $\mathcal{A}$*
  - $\texttt{Reveal}(\mathsf{S}_{j,t}, \texttt{main})$ *was never queried by $\mathcal{A}$, where $\mathsf{Partner}_i^s = j$, $\mathcal{M}_{j,t}^{\boldsymbol{in}} = \mathcal{M}_{i,s}^{\boldsymbol{out}}$, and $\mathcal{M}_{i,s}^{\boldsymbol{in}} = \mathcal{M}_{j,t}^{\boldsymbol{out}}$*

  - *the communication* $\mathsf{Partner}_i^s = j$ *is not* τ-corrupted *with* $\tau < \tau_0$*, where* $\mathtt{Test}(\mathsf{C}_{i,s}, \mathtt{main})$ *is the* τ_0*-th query issued by* $\mathcal{A}$
- *if* $\mathcal{A}$ *issues* $\mathtt{Test}(\mathsf{S}_{j,t}, \mathtt{tmp})$ *all of the following hold:*
  - $\mathtt{Reveal}(\mathsf{S}_{j,t}, \mathtt{tmp})$ *was never queried by* $\mathcal{A}$
  - *there exists an oracle* $\mathsf{C}_{i,s}$ *with* $\mathcal{M}_{i,s}^{\boldsymbol{out}} = \mathcal{M}_{j,t}^{\boldsymbol{in}}$
  - $\mathtt{Reveal}(\mathsf{C}_{i,s}, \mathtt{tmp})$ *was never queried by* $\mathcal{A}$ *where* $\mathsf{Partner}_j^t = i$ *and* $\mathcal{M}_{i,s}^{\boldsymbol{out}} = \mathcal{M}_{j,t}^{\boldsymbol{in}}$
  - $\mathtt{Reveal}(\mathsf{S}_{j,t'}, \mathtt{tmp})$ *was never queried by* $\mathcal{A}$ *for any* $\mathsf{S}_{j,t'}$ *with* $\mathcal{M}_{j,t}^{\boldsymbol{in}} = \mathcal{M}_{j,t'}^{\boldsymbol{in}}$
  - $\mathsf{S}_{j,}$ *is not* τ-corrupted *with* $\tau < \infty$
  - *the communication* $\mathsf{Partner}_j^t = i$ *is not* τ-corrupted *with* $\tau < \infty$
- *if* $\mathcal{A}$ *issues* $\mathtt{Test}(\mathsf{S}_{j,t}, \mathtt{main})$ *all of the following hold:*
  - $\mathtt{Reveal}(\mathsf{S}_{j,t}, \mathtt{main})$ *was never queried by* $\mathcal{A}$
  - *there exists an oracle* $\mathsf{C}_{i,s}$ *with* $\mathcal{M}_{i,s}^{\boldsymbol{out}} = \mathcal{M}_{j,t}^{\boldsymbol{in}}$
  - $\mathtt{Reveal}(\mathsf{C}_{i,s}, \mathtt{main})$ *was never queried by* $\mathcal{A}$*, where* $\mathsf{Partner}_j^t = i$ *and* $\mathcal{M}_{j,t}^{\boldsymbol{in}} = \mathcal{M}_{i,s}^{\boldsymbol{out}}$

else the game outputs a random bit. We define the advantage of $\mathcal{A}$ *to win the game* $\mathcal{G}_{\mathcal{A}}^{\mathcal{LLKE}-ma}$ *by*

$$\mathtt{Adv}_{\mathcal{A}}^{\mathcal{LLKE}-ma}(\lambda) := \left| \Pr[\mathcal{G}_{\mathcal{A}}^{\mathcal{LLKE}-ma}(\lambda) = 1] - \frac{1}{2} \right| .$$

Definition 9. *We say that a low-latency key exchange protocol under mutual authentication is* `test`*-secure if for all PPT adversaries* $\mathcal{A}$ *interacting according to the security game* $\mathcal{G}_{\mathcal{A}}^{\mathcal{LLKE}-sa}(\lambda)$ *it holds that*

$$\mathtt{Adv}_{\mathcal{A}}^{\mathcal{LLKE}-ma}(\lambda) \leq \mathsf{negl}(\lambda) .$$

6 Generic Construction of LLKE-M from NIKE

In this section we describe our generic construction of LLKE with mutual cryptographic authentication. Both the construction and its security analysis are very similar to their respective counterparts in the case of server-only authentication, the main differences are that now the message sent by the client is digitally signed, to authenticate the client. Accordingly, the security proof is adopted to the mutual-authentication case.

6.1 Generic Construction

Using the NIKE definition (2), we show how to generically construct a LLKE-M scheme from a NIKE scheme.

Let $\mathsf{NIKE} = (\mathsf{NIKEgen}, \mathsf{NIKEkey})$ be a NIKE scheme according to Definition 2 and let $\mathsf{SIGN} = (\mathsf{SIG.Gen}, \mathsf{SIG.Sign}, \mathsf{SIG.Vfy})$ be a signature scheme. Then we construct a LLKE-M scheme $\mathsf{LLKE\text{-}M} = (\mathsf{Gen}, \mathsf{M.KE}_{\mathsf{init}}^{\mathsf{client}}, \mathsf{M.KE}_{\mathsf{refresh}}^{\mathsf{client}}, \mathsf{M.KE}_{\mathsf{refresh}}^{\mathsf{server}})$, per Definition 7, in the following manner.

- $\mathsf{Gen}(1^\lambda, r)$ computes key pairs using the NIKE key generation algorithm $(pk^{\mathsf{nike}}, sk^{\mathsf{nike}}) \xleftarrow{\$} \mathsf{NIKEgen}(1^\lambda)$ and signature keys using the SIGN algorithm $(pk^{\mathsf{sg}}, sk^{\mathsf{sg}}) \xleftarrow{\$} \mathsf{SIG.Gen}$, and outputs
$$(pk, sk) := ((pk^{\mathsf{nike}}, pk^{\mathsf{sg}}), (sk^{\mathsf{nike}}, sk^{\mathsf{sg}})) \ .$$
- $\mathsf{M.KE}^{\mathsf{client}}_{\mathsf{init}}(pk_j, sk_i, r_i)$ samples $r_i \xleftarrow{\$} \{0,1\}^\lambda$, parses $(sk_i^{\mathsf{nike-static}}, sk_i^{\mathsf{sg}}))$ and $pk_j = (pk_j^{\mathsf{nike-static}}, pk_j^{\mathsf{sg}})$, and runs $(pk_i^{\mathsf{nike}}, sk_i^{\mathsf{nike}}) \leftarrow \mathsf{NIKEgen}(1^\lambda, r_i)$ and $k_{i,j}^{\mathsf{nike}} \leftarrow \mathsf{NIKEkey}(sk_i^{\mathsf{nike}}, pk_j^{\mathsf{nike-static}})$. It then computes $\sigma_i \leftarrow \mathsf{SIG.Sign}(sk_i^{\mathsf{sg}}, pk_i^{\mathsf{nike}})$ and outputs
$$(k_{\mathtt{tmp}}^{i,j}, m_i) := (k_{i,j}^{\mathsf{nike}}, (pk_i^{\mathsf{nike}}, \sigma_i)) \ .$$
- $\mathsf{M.KE}^{\mathsf{server}}_{\mathsf{refresh}}(sk_j, r_j, pk_i, m_i)$ parses $m_i = (pk_i^{\mathsf{nike}}, \sigma_i)$, $pk_i = (pk_i^{\mathsf{nike-static}}, pk_i^{\mathsf{sg}})$, and $sk_j = (sk_j^{\mathsf{nike-static}}, sk_j^{\mathsf{sg}})$, samples $r_j \xleftarrow{\$} \{0,1\}^\lambda$, and checks $\mathtt{true} \leftarrow \mathsf{SIG.Vfy}(pk_i^{\mathsf{sg}}, \sigma_i, pk_i^{\mathsf{nike}})$. It then computes $k_{i,j}^{\mathsf{nike}} \leftarrow \mathsf{NIKEkey}(sk_j^{\mathsf{nike-static}}, pk_i^{\mathsf{nike}})$, $(pk_j^{\mathsf{nike}}, sk_j^{\mathsf{nike}}) \leftarrow \mathsf{NIKEgen}(1^\lambda, r_j)$, $\sigma_j \leftarrow \mathsf{SIG.Sign}(sk_j^{\mathsf{sg}}, pk_j^{\mathsf{nike}})$, and $k_{\mathtt{main}}^{\mathsf{nike}} \leftarrow \mathsf{NIKEkey}(sk_j^{\mathsf{nike}}, pk_i^{\mathsf{nike}})$, and outputs
$$(k_{\mathtt{main}}^{j,i}, k_{\mathtt{tmp}}^{j,i}, m_j) := (k_{\mathtt{main}}^{\mathsf{nike}}, k_{i,j}^{\mathsf{nike}}, (pk_j^{\mathsf{nike}}, \sigma_j)) \ .$$
- $\mathsf{M.KE}^{\mathsf{client}}_{\mathsf{refresh}}(pk_j, sk_i, r_i, m_j)$ parses $pk_j = (pk_j^{\mathsf{nike-static}}, pk_j^{\mathsf{sg}})$ and $m_j = (pk_j^{\mathsf{nike}}, \sigma_j)$. It then checks $\mathtt{true} \leftarrow \mathsf{SIG.Vfy}(pk_j^{\mathsf{sg}}, \sigma_j, pk_j^{\mathsf{nike}})$ and computes $k_{\mathtt{main}}^{\mathsf{nike}} \leftarrow \mathsf{NIKEkey}(sk_i^{\mathsf{nike}}, pk_j^{\mathsf{nike}})$, and outputs
$$k_{\mathtt{main}}^{i,j} := k_{\mathtt{main}}^{\mathsf{nike}} \ .$$

6.2 Proof of Security for LLKE-M from NIKE Construction

Theorem 2. *From each attacker $\mathcal{A}$, we can construct attackers $\mathcal{B}_{sig}$, according to Definition 1, and $\mathcal{B}_{nike}$, according to Definition 3, such that*

$$\begin{aligned}
\mathtt{Adv}_{\mathcal{A}}^{\mathcal{LLKE}-ma}(\lambda) \leq \quad & dln \cdot \Big(\mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) + \mathtt{Adv}_{\mathsf{SIG},\mathcal{B}_{sig}}^{sEUF\text{-}CMA}(\lambda) \\
& \qquad + \mathtt{Adv}_{\mathsf{SIG},\mathcal{B}_{csig}}^{sEUF\text{-}CMA}(\lambda) \Big) \\
& + dln \cdot \Big(k \cdot \mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) + \mathtt{Adv}_{\mathsf{SIG},\mathcal{B}_{sig}}^{sEUF\text{-}CMA}(\lambda) \\
& \qquad + \mathtt{Adv}_{\mathsf{SIG},\mathcal{B}_{csig}}^{sEUF\text{-}CMA}(\lambda) \Big) \\
& + 2kdln \cdot \Big(\mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) + \mathtt{Adv}_{\mathsf{SIG},\mathcal{B}_{sig}}^{sEUF\text{-}CMA}(\lambda) \\
& \qquad + \mathtt{Adv}_{\mathsf{SIG},\mathcal{B}_{csig}}^{sEUF\text{-}CMA}(\lambda) \Big) \\
& + 4 \cdot \mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) \ .
\end{aligned}$$

The running time of $\mathcal{B}_{sig}$ and $\mathcal{B}_{nike}$ is approximately equal to the running time of $\mathcal{A}$ for the simulation of the security experiment for $\mathcal{A}$.

Proof Sketch. Again we distinguish between four attackers:

- adversary $\mathcal{A}_5$ asks `Test`() to a client oracle and the temporary key (*CT-attacker*)
- adversary $\mathcal{A}_6$ asks `Test`() to a client oracle and the main key (*CM-attacker*)
- adversary $\mathcal{A}_7$ asks `Test`() to a server oracle and the temporary key (*ST-attacker*)
- adversary $\mathcal{A}_8$ asks `Test`() to a server oracle and the main key (*SM-attacker*)

From these, Lemmas 5-8 complete the proof of Theorem 2.

Lemma 5. *From each* CT-attacker $\mathcal{A}_5$*, we can construct attackers* $\mathcal{B}_{sig}$ *and* $\mathcal{B}_{csig}$*, according to Definition 1, and* $\mathcal{B}_{nike}$*, according to Definition 3, such that*

$$\begin{aligned}\mathtt{Adv}^{\mathcal{LLKE}-ma}_{\mathcal{A}_5}(\lambda) \leq \quad & dln \cdot \Big(\mathtt{Adv}^{\textit{CKS-light}}_{\mathsf{NIKE},\mathcal{B}_{nike}}(\lambda) + \mathtt{Adv}^{\textit{sEUF-CMA}}_{\mathsf{SIG},\mathcal{B}_{sig}}(\lambda) \\ & + \mathtt{Adv}^{\textit{sEUF-CMA}}_{\mathsf{SIG}}(\mathcal{B}_{csig})\Big) + \mathtt{Adv}^{\textit{CKS-light}}_{\mathsf{NIKE},\mathcal{B}_{nike}}(\lambda) \ .\end{aligned}$$

The running time of $\mathcal{B}_{sig}$ and $\mathcal{B}_{nike}$ is approximately equal to the running time of $\mathcal{A}_5$ for the simulation of the security experiment for $\mathcal{A}_5$.

Proof Sketch. As the proof of for LLKE-M follows closely to that of Lemma 1, we highlight the main differences for conciseness.

- In Lemma 1, Game 2, we additionally choose a random $s' \xleftarrow{\$} \{1 \dots n\}$, aborting the experiment if the attacker $\mathcal{A}_5$ does not ask the `Test`-query to $C_{i,s}$ for $i = i'$ and $s = s'$.
- After Lemma 1, Game 4 we add the additional condition we abort the game and output a random bit if $\mathcal{M}^{\mathbf{in}}_{j,t} = (pk^{\mathsf{nike}}_{i,s}, \sigma_i)$ where $\mathtt{true} \leftarrow \mathsf{SIG.Vfy}(pk^{\mathsf{sg}}_i, \sigma_i, pk^{\mathsf{nike}}_{i,s})$ and no oracle of i has previously computed the pair $(pk^{\mathsf{nike}}_{i,s}, \sigma_i)$. Now the game is indistinguishable from Lemma 1, Game 4, else it would be possible to build an attacker $\mathcal{B}_{csig}$ that could generate a valid client signature.
- $\mathcal{B}_{nike}$ continues to simulate the experiment as in the proof of Lemma 1.
  - $\mathcal{A}$ may ask Corrupt-queries for any server oracle $\mathsf{S}_{j',t}$, such that $j \neq j'$, and any client oracle $\mathsf{C}_{i'}s$, such that $i \neq i'$.

  `Reveal`- and `Send`-queries are handled as in the proof of Lemma 1.

Lemma 6. *From each* CM-attacker $\mathcal{A}_6$*, we can construct attackers* $\mathcal{B}_{sig}$*, according to Definition 1, and* $\mathcal{B}_{nike}$*, according to Definition 3, such that*

$$\begin{aligned}\mathtt{Adv}^{\mathcal{LLKE}-ma}_{\mathcal{A}_6}(\lambda) \leq \quad & dln \cdot \Big(k \cdot \mathtt{Adv}^{\textit{CKS-light}}_{\mathsf{NIKE},\mathcal{B}_{nike}}(\lambda) + \mathtt{Adv}^{\textit{sEUF-CMA}}_{\mathsf{SIG},\mathcal{B}_{sig}}(\lambda) \\ & + \mathtt{Adv}^{\textit{sEUF-CMA}}_{\mathsf{SIG}}(\mathcal{B}_{csig})\Big) + \mathtt{Adv}^{\textit{CKS-light}}_{\mathsf{NIKE},\mathcal{B}_{nike}}(\lambda) \ .\end{aligned}$$

The running time of $\mathcal{B}_{sig}$ and $\mathcal{B}_{nike}$ is approximately equal to the running time of $\mathcal{A}_6$ for the simulation of the security experiment for $\mathcal{A}_6$.

Notably, the main differences between the proofs of Lemma 2 and Lemma 6 are exactly those described in the proof sketch of Lemma 5 above, with the following addition.

$\mathcal{A}$ may ask `Corrupt`-queries on any server oracle $\mathsf{S}_{j',t}$, such that $j \neq j'$, and any client oracle $\mathsf{C}_{i's}$, such that $i \neq i'$. However, it may also ask its τ-th query as a `Corrupt`-query on $\mathsf{S}_{j,t}$ or $\mathsf{C}_{i,s}$ if $\tau > \tau_0$, where $\mathtt{Test}(\mathsf{S}_{j,t}, \mathtt{main})$ was the τ_0-th query.

Lemma 7. *From each* ST-attacker $\mathcal{A}_7$*, we can construct attackers* $\mathcal{B}_{sig}$*, according to Definition 1, and* $\mathcal{B}_{nike}$*, according to Definition 3, such that*

$$\begin{aligned}\mathtt{Adv}_{\mathcal{A}_7}^{\mathcal{LLKE}-ma}(\lambda) \leq \quad & kdln \cdot \Big(\mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) + \mathtt{Adv}_{\mathsf{SIG},\mathcal{B}_{sig}}^{sEUF\text{-}CMA}(\lambda) \\ & + \mathtt{Adv}_{\mathsf{SIG}}^{sEUF\text{-}CMA}(\mathcal{B}_{csig})\Big) + \mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) \ .\end{aligned}$$

The running time of $\mathcal{B}_{sig}$ *and* $\mathcal{B}_{nike}$ *is approximately equal to the running time of* $\mathcal{A}_7$ *for the simulation of the security experiment for* $\mathcal{A}_7$*.*

Proof Sketch. Here again, the proof follows that of Lemma 3, adapted as in that of Lemma 5.

Lemma 8. *From each* SM-attacker $\mathcal{A}_8$*, we can construct attackers* $\mathcal{B}_{sig}$*, according to Definition 1, and* $\mathcal{B}_{nike}$*, according to Definition 3, such that*

$$\begin{aligned}\mathtt{Adv}_{\mathcal{A}_8}^{\mathcal{LLKE}-ma}(\lambda) \leq \quad & kdln \cdot \Big(\mathtt{Adv}_{\mathsf{SIG},\mathcal{B}_{sig}}^{sEUF\text{-}CMA}(\lambda) + \mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) \\ & + \mathtt{Adv}_{\mathsf{SIG}}^{sEUF\text{-}CMA}(\mathcal{B}_{csig})\Big) + \mathtt{Adv}_{\mathsf{NIKE},\mathcal{B}_{nike}}^{CKS\text{-}light}(\lambda) \ .\end{aligned}$$

The running time of $\mathcal{B}_{sig}$ *and* $\mathcal{B}_{nike}$ *is approximately equal to the running time of* $\mathcal{A}_8$ *for the simulation of the security experiment for* $\mathcal{A}_8$*.*

Here the proof follows according to that of Lemma 4, edited according to the description in the proof sketch of Lemma 5. Additionally, in the experiment simulation by $\mathcal{B}_{nike}$, $\mathcal{A}$ may ask `Corrupt`-queries on any server oracle $\mathsf{S}_{j',t}$, such that $j \neq j'$, and any client oracle $\mathsf{C}_{i's}$, such that $i \neq i'$. However, it may also ask its τ-th query as a `Corrupt`-query on $\mathsf{S}_{j,t}$ or $\mathsf{C}_{i,s}$ if $\tau > \tau_0$, where $\mathtt{Test}(\mathsf{S}_{j,t}, \mathtt{main})$ was the τ_0-th query.

References

1. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93*, pages 62–73, Fairfax, Virginia, USA, November 3–5, 1993. ACM Press.
2. Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 232–249, Santa Barbara, CA, USA, August 22–26, 1994. Springer, Heidelberg, Germany.

3. Florian Bergsma, Tibor Jager, and Jörg Schwenk. One-round key exchange with strong security: An efficient and generic construction in the standard model. In Jonathan Katz, editor, *PKC 2015*, volume 9020 of *LNCS*, pages 477–494, Gaithersburg, MD, USA, March 30 – April 1, 2015. Springer, Heidelberg, Germany.
4. Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella Béguelin. Proving the TLS handshake secure (as it is). In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 235–255, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.
5. Dan Boneh, Emily Shen, and Brent Waters. Strongly unforgeable signatures based on computational Diffie-Hellman. In Yung et al. [23], pages 229–240.
6. Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 453–474, Innsbruck, Austria, May 6–10, 2001. Springer, Heidelberg, Germany.
7. David Cash, Eike Kiltz, and Victor Shoup. The twin Diffie-Hellman problem and applications. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 127–145, Istanbul, Turkey, April 13–17, 2008. Springer, Heidelberg, Germany.
8. Cas Cremers. Examining indistinguishability-based security models for key exchange protocols: the case of CK, CK-HMQV, and eCK. In Bruce S. N. Cheung, Lucas Chi Kwong Hui, Ravi S. Sandhu, and Duncan S. Wong, editors, *ASIACCS 11*, pages 80–91, Hong Kong, China, March 22–24, 2011. ACM Press.
9. Marc Fischlin and Felix Günther. Multi-stage key exchange and the case of Google's QUIC protocol. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 1193–1204, Scottsdale, AZ, USA, November 3–7, 2014. ACM Press.
10. Eduarda S. V. Freire, Dennis Hofheinz, Eike Kiltz, and Kenneth G. Paterson. Non-interactive key exchange. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 254–271, Nara, Japan, February 26 – March 1, 2013. Springer, Heidelberg, Germany.
11. 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, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.
12. Hugo Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 546–566, Santa Barbara, CA, USA, August 14–18, 2005. Springer, Heidelberg, Germany.
13. Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 429–448, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.
14. Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *ProvSec 2007*, volume 4784 of *LNCS*, pages 1–16, Wollongong, Australia, November 1–2, 2007. Springer, Heidelberg, Germany.
15. Kristin Lauter and Anton Mityagin. Security analysis of KEA authenticated key exchange protocol. In Yung et al. [23], pages 378–394.
16. Laurie Law, Alfred Menezes, Minghua Qu, Jerry Solinas, and Scott Vanstone. An efficient protocol for authenticated key agreement. *Designs, Codes and Cryptography*, 28(2):119–134, 2003.

17. Robert Lychev, Samuel Jero, Alexandra Boldyreva, and Cristina Nita-Rotaru. How secure and quick is QUIC? provable security and performance analyses. In *2015 IEEE Symposium on Security and Privacy*, pages 214–231, San Jose, California, USA, May 17–21, 2015. IEEE Computer Society Press.
18. NIST. Skipjack and kea algorithm specifications, 1998. `http://csrc.nist.gov/groups/STM/cavp/documents/skipjack/skipjack.pdf`.
19. E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3: draft-ietf-tls-tls13-08. Technical report, August 2015. Expires February 29, 2016.
20. Augustin P. Sarr, Philippe Elbaz-Vincent, and Jean-Claude Bajard. A new security model for authenticated key agreement. In Juan A. Garay and Roberto De Prisco, editors, *SCN 10*, volume 6280 of *LNCS*, pages 219–234, Amalfi, Italy, September 13–15, 2010. Springer, Heidelberg, Germany.
21. Ron Steinfeld, Josef Pieprzyk, and Huaxiong Wang. How to strengthen any weakly unforgeable signature into a strongly unforgeable signature. In Masayuki Abe, editor, *CT-RSA 2007*, volume 4377 of *LNCS*, pages 357–371, San Francisco, CA, USA, February 5–9, 2007. Springer, Heidelberg, Germany.
22. Kazuki Yoneyama and Yunlei Zhao. Taxonomical security consideration of authenticated key exchange resilient to intermediate computation leakage. In Xavier Boyen and Xiaofeng Chen, editors, *ProvSec 2011*, volume 6980 of *LNCS*, pages 348–365, Xi'an, China, October 16–18, 2011. Springer, Heidelberg, Germany.
23. Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors. *PKC 2006*, volume 3958 of *LNCS*, New York, NY, USA, April 24–26, 2006. Springer, Heidelberg, Germany.