Proving Authenticated Key Exchange via Memory-Tight Reductions

Jiaxin Pan[⋆] and Runzhi Zeng

University of Kassel, Kassel, Germany
{jiaxin.pan, runzhi.zeng}@uni-kassel.de

Abstract. We initiate the study of memory efficiency in proving the security of authenticated key exchange (AKE) protocols: We first revise the security model for AKE protocols in order to prove their security in a memory-efficient manner without compromising its capability of capturing usual attacks. We formally show that security in our model implies security in previous ones, and thus our model captures the same security as before.

After that we propose a generic construction of AKE from key encapsulation mechanisms (KEMs) and digital signature schemes, motivated by the signed Diffie-Hellman protocol. Under the multi-user security of the signature scheme and (relatively weak) one-way security against plaintext checking attacks of the KEM, our generic construction is proven to be tightly secure (in terms of success probability) via *memory-tight* reductions in the random oracle model. This gives us the *first* memory-tight AKE protocol, and it largely reduces the memory consumption of proving AKE protocols. Given that most post-quantum assumptions (e.g., the Learning-With-Errors and Short-Integer-Solution assumptions) are memory-sensitive, our improvement on memory consumption holds significant value for post-quantum AKE protocols.

Keywords: Authenticated key exchange, memory tightness, random oracles.

1 Introduction

Reductions are crucial for proving security of cryptography, where a reduction is constructed to show that if an adversary $\mathcal{A}$ can break the security of a protocol Π then the reduction $\mathcal{R}$ can use $\mathcal{A}$ to solve some underlying hard problem P. By the hardness of these well-established problems, the security of Π is concluded.

Concrete Security and Time-Advantage-Tightness. The analysis of concrete security is to quantify security by measuring how many resources the reduction $\mathcal{R}$ needs in addition to those of $\mathcal{A}$. We often call this tightness between a security reduction and an adversary. Previous work (e.g. [BR96,BR09,HJ12,PQR22]) typically considers the Time-Advantage (TA) tightness between adversary $\mathcal{A}$ and the corresponding reduction $\mathcal{R}$. More precisely, let $(t_{\mathcal{A}}, \varepsilon_{\mathcal{A}})$ and $(t_{\mathcal{R}}, \varepsilon_{\mathcal{R}})$ be the running time and advantage of $\mathcal{A}$ and $\mathcal{R}$, respectively, and they show that $t_{\mathcal{A}} \approx t_{\mathcal{R}}$ and $\varepsilon_{\mathcal{A}} \leq L \cdot \varepsilon_{\mathcal{R}}$, where L is called the security loss of a reduction. A smaller L gives a tighter reduction and stronger security guarantees for a cryptographic protocol. When L is a small constant, the security reduction is called tight in the early literature (such as [BR96,BR09,HJ12,DGJL21b,PQR22]), and we write $\varepsilon_{\mathcal{R}} \approx \varepsilon_{\mathcal{A}}$.

Achieving tightness is highly desirable, since it does not only provide stronger security guarantees as mentioned earlier, but also allows one to instantiate the scheme with smaller security parameters. In the past 10 years, it has been an active direction to study tightly secure cryptographic primitives, such as signature schemes [HJ12,DGJL21b,PW22], public-key encryption schemes [GHKW16,HLG23], and authenticated key exchange (AKE) protocols [BHJ+15,GJ18,LLGW20,JKRS21,PQR22,PWZ23a].

Memory-Tight / Memory-Efficient Reductions. Memory efficiency is a crucial aspect to consider when analyzing algorithms, yet it has often been neglected in security reductions. Memory inefficiency can undermine the effectiveness of tight security reductions, especially since several problems commonly used in cryptography are sensitive to memory capacity. In essence, these problems can be resolved more swiftly with access to larger memory resources. We refer to [ACFK17, Section 6] for a discussion of memory-sensitive problems and their time-memory tradeoffs. According to the best known cryptanalytic

⋆ Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 572525990

results [LLMP93,Pol75,EKM17,BKW03,CN11,APS15,HKM18,BDGL16,HK17], prominent examples in the public-key setting include factoring, Discrete Logarithms (DL) in finite fields, Learning Parities with Noise (LPN), Learning With Errors (LWE), approximate Shortest Vector Problem, and Short Integer Solution (SIS).

Surprisingly, many known tightly secure cryptosystems are based on memory-sensitive problems, but their security reductions are not memory-efficient, for instance, the tight LWE-based AKE protocol of Pan, Wagner, and Zeng [PWZ23a]. These schemes often need to maintain several lists to keep track of adversaries' queries and to program the random oracles (ROs). In 2017, Auerbach, Cash, Fersch, and Kiltz [ACFK17] initiated the analysis of memory efficiency for security reductions and proposed the notion of memory-tight reductions, meaning $m_{\mathcal{R}} \approx m_{\mathcal{A}}$ where $m_{\mathcal{A}}$ and $m_{\mathcal{R}}$ are the amounts of memory used by $\mathcal{A}$ and $\mathcal{R}$. Following [JK22], we distinguish the two different tightness notions, Time-Advantage (or TA for short) tightness and Time-Advantage-Memory (or TAM) tightness, in this section.

Since its introduction in 2017, memory tightness is a well-established concept that attracts broad interest, but only limited types of primitives can be constructed via TAM-tight reductions. These are the RSA- and Diffie-Hellman-based (DH-based) signature schemes [DGJL21a], the DH-based public-key encryption schemes [Bha20,JK22,GGJT22], and the "Encrypt-then-PRF" authenticated encryption scheme [GGJT22]. This indicates that constructing TAM-tight schemes is rather challenging, since a TAM-tight reduction has to simulate the security game with limited memory and time. In some cases [ACFK17,WMHT18,GJT20,GT20] it is even proven to be impossible. Hence, it is interesting not only in practice, but also in theory, to investigate the memory efficiency of proving primitives beyond signature and encryption schemes.

To illustrate the impact of memory loss, consider a hypothetical memory-sensitive problem whose estimated attack cost, at a fixed success probability, is $2^{0.8n}$ time with memory below $2^{0.2n}$, but only $2^{0.3n}$ time with memory $2^{0.2n}$. For an adversary with fixed memory, we choose n large enough to keep its memory below this threshold. With a memory-tight reduction, we can keep the reduction's memory below the threshold as well and use the $2^{0.8n}$ attack cost in the security estimate. However, a memory-inefficient reduction may exceed this threshold even when the adversary does not. We must then either increase n to keep the reduction below the threshold or account for the faster $2^{0.3n}$ attack in the security estimate.

This loss in the security guarantee is in addition to the loss over time and advantage. Therefore, memory-tightness is also useful for non-TA-tight reductions from memory-sensitive assumptions in this sense: even when their TA loss remains, a memory-tight reduction avoids the additional loss caused by its memory overhead.

Our Goal: Proving AKE via Memory-Efficient Reductions. Our paper focuses on the memory efficiency of proving security of AKE protocols. This is particularly important for post-quantum protocols, as major post-quantum assumptions (e.g. LWE and SIS) are memory-sensitive.

An AKE protocol allows two parties to agree on a session key for a secure channel, and AKE protocols are crucial for secure communication over the internet. TA-tight AKE protocols are often constructed from the DH assumption [BHJ+15,GJ18,JKRS21,PQR22] or the LWE assumption [PWZ23a]. The security reductions of these protocols consume a large amount of memory and are highly memory-inefficient. This may not pose a significant problem for DH-based protocols, since the DL problem over elliptic curves is not memory-sensitive. However, for lattices, the large memory consumption lowers the security guarantee of the LWE-based AKE protocol [PWZ23a], even though it is already TA-tight, since the LWE problem is memory-sensitive.

We stress that the memory-tightness is interesting, in particular, for non-TA-tight AKE protocols (e.g. [HKSU20,PWZ23b]) from memory-sensitive assumptions, since these protocols suffer not only from security loss over time and advantage, but also from memory-inefficient reductions that may further diminish their security guarantees. Currently, there is no lattice-based AKE protocol (regardless of its TA-tightness) with memory-efficient reductions.

Difficulties in Achieving the Goal. For AKE, it is harder to construct reductions with low memory usage than for encryption or signatures, since AKE attacks are more complex and dynamic. Adversaries against AKE have full control over the communication channel, and they can reveal the shared session keys and corrupt some of the honest users' long-term secrets. As a result, the AKE security model needs to keep track of the protocol transcripts that have been transferred, the shared session keys, and the honest users' long-term keys. This requires a large amount of memory. In addition, for protocols in the random oracle (RO) model, one will need to keep track of the RO history. Hence, the security proof of

AKE protocols is rather memory-inefficient. Concretely, the memory consumption of proving TA-tight AKE protocols in [GJ18,JKRS21,PQR22,PWZ23a] is roughly

$$m_{\mathcal{A}} + (\mathrm{O}(N_{\mathtt{user}}) + \mathrm{O}(N_{\mathtt{ses}}) + \mathrm{O}(Q_{\mathsf{RO}})) \cdot m', \tag{1}$$

where $N_{\mathtt{user}}$, $N_{\mathtt{ses}}$, and Q_{RO} are numbers of users, sessions, and RO queries, respectively, and m' is the bit-length of representing either a group element for the DH-based protocols or a lattice vector for the lattice-based one. For today's social media applications, $N_{\mathtt{user}}$ and $N_{\mathtt{ses}}$ can easily be billions, namely, $(N_{\mathtt{user}}, N_{\mathtt{ses}}) \approx (2^{32}, 2^{48})$. Under these numbers, Q_{RO} can be up to 2^{48} (which is relevant to the number of sessions). Such a large memory consumption as in Equation (1) lowers the security guarantees provided by their TA-tight proofs and restricts the interest of TA-tight AKE.

Not only do the TA-tight AKE protocols suffer from memory-inefficient reductions, but so do the non-TA-tight ones. We consider the non-TA-tight ones in [FSXY12,HKSU20,PWZ23b], since they can be implemented with post-quantum assumptions. The memory consumption of their security reductions is at least

$$m_{\mathcal{A}} + (\mathrm{O}(N_{\mathtt{user}}) + \mathrm{O}(N_{\mathtt{ses}})) \cdot m'. \tag{2}$$

AKE protocols in the quantum RO model (QROM) [HKSU20,PWZ23b] are proven secure with reductions that do not reprogram the RO, since a quantum adversary can query a RO with a superposition, and it is hard to program this. Hence, it removes the $\mathrm{O}(Q_{\mathsf{RO}})$-term in Equation (1). Similarly, it is the case for the protocol in the standard model [FSXY12]. However, $N_{\mathtt{ses}}$ remains a dominant factor.

1.1 Our Contributions

We improve the memory efficiency for proving AKE protocols. Specifically,

- We propose a new AKE security model that allows us to record adversaries' history in a memory-efficient manner, and
- We construct a generic AKE protocol that preserves the TA-tightness of the underlying primitives with *memory-tight* proofs. Our construction requires signatures and key encapsulation mechanisms (KEMs). Different to an ad-hoc protocol, ours is flexible and can be efficiently implemented with various memory-sensitive assumptions (e.g. LWE or DH over integers). We stress that this protocol is the first memory-tight AKE protocol.

Technical Overview. We detail our contributions as follows:

A New Model. The main motivation of a new model is that the current AKE models may not be suitable for memory-tight reductions. For instance, the models in [JKRS21,PWZ23a,GGJJ23,PWZ23b] need to record information about whether a session exists, and this session has either concluded a session key or been rejected. As a result, security reductions in these models have to create extra memory and store this information. Our new model modifies the way of maintaining session information. For the formal comparison, we use a single-Test model adapted from Pan, Wagner, and Zeng [PWZ23b]. Formally, we show (cf. Theorem 2 in Supp. Mat. C) that if a protocol is secure in our new model, then it is also secure in this adapted comparison model. Note that the comparison reduction is not memory-tight, since its purpose is to relate security in our model to security in the adapted single-Test game. We suppose that this raises up the question of what is the "right" security model for AKE. For instance, our model is more preferable for protocols based on memory-sensitive assumptions, while the previous ones are for those with non-memory-sensitive assumptions. We consider two-message AKE models because our memory-efficient protocol is two-message. Our model is presented in Section 3.

A Generic AKE with Memory-tight Proofs. We propose an AKE protocol *generically* from digital signatures and KEMs. In our new security model, we show that our protocol preserves the TA-tightness of the signature and KEM schemes with *memory-tight* reductions. This is in stark contrast to the existing proofs for AKE, as we largely improved the reductions' memory-consumption as described in Equation (1). Our proof is in the random oracle model and requires the multi-user security of the signature scheme and the multi-user, one-way security against plaintext checking attacks (OW-PCA) of the KEM.

In a nutshell, our generic protocol can be viewed as a memory-efficient generalization of the two-message Signed Diffie-Hellman protocol (as in [PQR22]) with a subtle modification. The modification is to expand a random nonce in each protocol message, and the nonce enables our reduction to simulate the security game memory-efficiently. Although we add the nonce into the protocol transcript, this overhead is rather small, in particular, when we implement it with lattices. Compared with Kyber ciphertexts (≈ 1 KB) and Dilithium signatures (≈ 2.4 KB), our nonce is about 64 Bytes, and this is marginal.

We briefly sketch some technical details here: We recall that the signed DH protocol is to use a digital signature scheme to sign the unauthenticated DH protocol. In 2022, Pan et al. [PQR22] showed that the signed DH protocol is TA-tightly secure in the random oracle model, but their proof is highly memory-loose as in Equation (1). Our approach is to keep the TA-tightness and make it memory-tight. We further generalize it with a suitable abstraction that can be implemented with post-quantum assumptions, such as LWE. For simplicity, we sketch our ideas with the DH terminology.

The proof in [PQR22] has two major steps: Firstly, they eliminate active attacks with a TA-tight reduction to the signature security. It requires the signature scheme to be secure in the multi-user setting against adaptive signing key corruptions. We note that existing memory-tight signatures [DGJL21a,Xag24] do not consider adaptive corruptions, and we will discuss how to construct a new scheme for it with memory-tight reductions.

Now, since the active attacks have been eliminated, the reduction in [PQR22] can randomize the computational DH (CDH) challenge and embed it into every session TA-tightly by the random self-reducibility of CDH. Although this reduction cannot compute the session key for any session, it can use the strong DH predicate [ABR01] to carefully patch the random oracle so that the AKE adversary $\mathcal{A}$ does not detect that. After doing so, $\mathcal{A}$ has to solve the CDH challenge. This second step requires the use of the strong DH assumption [ABR01]. Because of patching the random oracle (RO), the reduction in [PQR22] needs to use its memory to keep track of the RO history. Moreover, it has to store the protocol transcripts for each session and the user secret keys to answer corruption Cor queries.

To remove the storage for RO queries, an idea is to replace the RO with a pseudo-random function (PRF) as in [ACFK17]. However, this is not straightforward because the reduction itself cannot compute the session keys for sessions where the DH challenges are embedded, and it needs to carefully patch the RO to "hide" this fact. Also, storing the protocol transcripts seems necessary for the reduction in [PQR22] to check if the adversary submits a valid attack or not.

To overcome these two fundamental obstacles, we slightly modify the signed DH protocol by introducing a random nonce in each protocol transcript. We develop a message encoding technique that embeds critical information into these nonces. This allows our reduction to avoid storing the protocol transcripts. Instead, we extract the transcripts from the adversary's queries only when needed to simulate RO queries or the security game. As a result, we remove the memory consumption of storing RO queries and protocol transcripts. This leads to a memory-tight reduction, while preserving the TA-tightness. Our next question is whether the underlying KEM and signature schemes can be constructed with TA- and memory-tight (aka. TAM-tight) reductions.

TAM-Tight KEMs with Implicit Rejection. In our AKE proof, we require the KEM to be one-way secure against plaintext checking attacks (OW-PCA) in the multi-user, multi-challenge setting, where an adversary can see multiple public keys, check whether an encapsulated key is valid wrt a ciphertext and some user, and obtain multiple challenge ciphertexts wrt different users. This can be viewed as an abstraction of how the strong DH assumption is used in [PQR22]. Unlike the signature scheme (which will be discussed later), we do not require security against adaptive corruptions for the KEM. We show that a multi-user, multi-challenge IND-CPA-secure PKE can TAM-tightly imply our multi-user, multi-challenge OW-PCA-secure KEM. The IND-CPA-secure PKE can be implemented with LWE. Our technique is derived from a combination of the QROM techniques in [JZC+18,SXY18] and methods from memory-tight PKE constructions in [Bha20,JK22].

We note that Jaeger et al. [JK22] turn an IND-CPA PKE into an OW-PCA PKE. Although their results can also be adapted to instantiate our AKE protocol and have similar security bounds as ours, their constructions consider explicit rejection, which incurs extra ciphertext expansion. Instead, our KEM has implicit rejection, which is sufficient for our AKE constructions and does not have extra ciphertext expansion and thus is more efficient. More details are given in Remark 6.

Memory-Tight Signatures with Adaptive Corruptions. Our AKE construction requires a multi-user-secure signature scheme, where an adversary $\mathcal{A}$ receives multiple users' public keys and adaptively corrupts some user signing keys, and it can furthermore ask for a signature for some message m under user u. To win, $\mathcal{A}$ has to submit a signature σ^* on some message m^* under user u^*. We require strong existential unforgeability, namely, an adversary can forge on a previous message-user pair (m^*, u^*) with a fresh signature σ^*. Moreover, our AKE proofs only require the weaker one-per-message security as in [FKP17], where the signing oracle only issues one signature per message-user pair. We name our security

notion as multi-user strong unforgeability against adaptive corruptions and chosen-one-per-message attack (aka. MU^c-sEUF-1CMA).

We show that the single-user variant of our notion (sEUF-1CMA) implies MU^c-sEUF-1CMA with a memory-tight reduction, but the reduction loses a factor of the number of public keys in terms of advantage. Combining with the previous work in [DGJL21a,Xag24], this gives us a memory-tight, TA-non-tight signature scheme against adaptive corruptions from various assumptions, including lattices. While the previous memory-tight schemes [DGJL21a,Xag24] do not consider adaptive corruptions, our scheme is the first achieving memory-tightness in this setting, although it is TA-non-tight. We leave constructing TAM-tight signatures against adaptive corruptions as an open problem.

Instantiations. Putting things together, we can instantiate our generic AKE construction with integer lattices (using the LWE and SIS assumptions) or module lattices (using the module-LWE (MLWE) and module-SIS (MSIS) assumptions). Both instantiations are memory-tight, namely, the reductions' memory consumption is independent of the number of users $N_{\mathtt{user}}$. For integer lattices, we manage to achieve "almost" TA-tightness, where the advantage loses a factor of $\mathrm{O}(N_{\mathtt{user}}\lambda)$ where λ is the security parameter. This loss is due to the MU^c-sEUF-1CMA signature scheme. The work of Pan, Wagner, and Zeng [PWZ23b] has the same TA-tightness as ours, but our proof is memory-tight (cf. the memory complexity of [PWZ23b] as in Equation (2)). More details of our instantiations can be found in Supp. Mat. A.

Limitations on Our Work. Our AKE proof preserves the TA- and memory-tightness of the underlying KEM and signature scheme, and we already showed that the KEM can be constructed in a TAM-tight manner, but it seems inherently difficult to construct a TAM-tight signature scheme with adaptive corruptions. We sketch the technical difficulties here, and it may serve as a starting point for future work.

Technically, with adaptive corruptions, a reduction has to decide whether a forgery belongs to a non-corrupted user. Otherwise, the forgery is not necessarily a valid attack on the signature scheme. (One may think of using adaptive random oracle programming to avoid this, but such programming itself is not memory-tight.) Hence, it seems unavoidable to record the corrupted users. To provide more details, we sketch a simple argument on it. Deciding whether a user has been corrupted can be understood as the following *dictionary* problem: Given a set $S \subset \{1, ..., M\}$ and $|S| = N$, decide if a given x is in S, where S can be adversarial, namely, given by an adversary and not known to the algorithm in advance. Any algorithm solving this problem without error needs to store the whole set S, otherwise one may query an element that belongs to S but has not been recorded, which may cause the algorithm to output an incorrect answer. Since there are $\binom{M}{N}$ options for S, Pagh [Pag02] shows that storing such a set requires at least $\log(\binom{M}{N})$ bits and

$$\log(\binom{M}{N}) = N\log(\frac{M}{N}) - \Theta(\frac{N^2}{M}) - \mathrm{O}(\log N).$$

In our case, $M = |\mathcal{U}| = 2^{O(\lambda)}$ corresponds to the size of the space of user identifiers and $N = N_{\mathtt{user}}$ the number of users. Hence, having a memory-loss depending on $N_{\mathtt{user}}$ seems unavoidable when constructing a TAM-tight signature scheme with adaptive corruptions.

2 Preliminaries

For an integer n, $[n] := \{1, \ldots, n\}$ and $(n] := \{0, 1, \ldots, n\}$. Let $\mathcal{X}$ and $\mathcal{Y}$ be two finite sets. We define $x \leftarrow \mathcal{X}$ to denote sampling x from $\mathcal{X}$ uniformly at random. $\mathsf{Func}(\mathcal{X}, \mathcal{Y})$ is the set of all functions with domain $\mathcal{X}$ and codomain $\mathcal{Y}$.

Let $\mathcal{A}$ be an algorithm. We write $y \leftarrow \mathcal{A}(x)$ to denote that $\mathcal{A}$ outputs y on input x. If $\mathcal{A}$ is deterministic, then we may write $y := \mathcal{A}(x)$. We also write $y \leftarrow$ (or $:=$) $\mathcal{A}^f(x)$ if $\mathcal{A}$ has black-box access to some function (or oracle) f.

We use code-based games [BR06] to define and prove security. For all security games, we assume that all Boolean flags are initialized to false and sets and (ordered) lists are initialized to $\emptyset$. Let $\mathcal{L}$ be an ordered list. If $\mathcal{L}[i]$ has not been assigned a value, then we define $\mathcal{L}[i] = \bot$. The notation $\Pr[\mathbf{G}^{\mathcal{A}} \Rightarrow 1]$ denotes the probability that the final output of game $\mathbf{G}$ running an adversary $\mathcal{A}$ is 1.

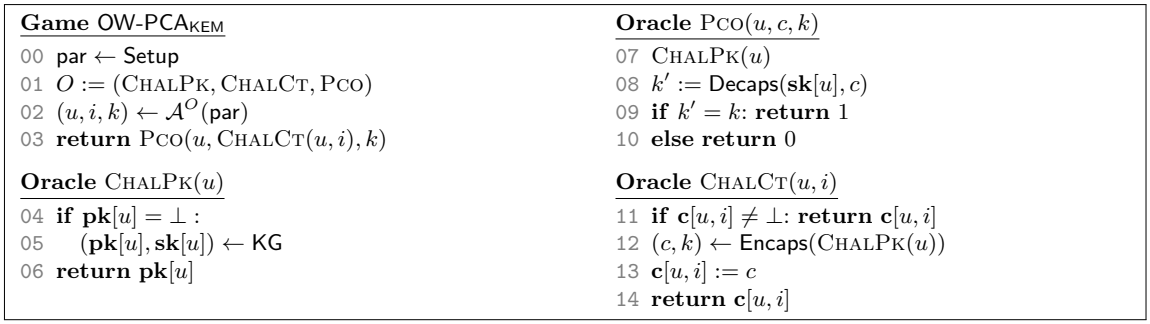

Fig. 1. Security game for defining OW-PCA security. We implicitly assume that all user identifiers belong to $\mathcal{U}$.

2.1 Definitions of Key Encapsulation Mechanisms

A KEM scheme KEM consists of four algorithms (Setup, KG, Encaps, Decaps) and a ciphertext space $\mathcal{C}$, a randomness space $\mathcal{R}$, and a KEM key space $\mathcal{K}$:

- On input security parameter λ, Setup outputs a system parameter par. The key generation algorithm KG(par) outputs a public-secret key pair (pk, sk). Since the security parameter and system parameter are public, for simplicity, we do not write them in the algorithms' inputs explicitly.
- The encapsulation algorithm Encaps, on input pk, outputs a ciphertext $c \in \mathcal{C}$ and a KEM key $k \in \mathcal{K}$.
- The decapsulation algorithm Decaps, on input sk and a ciphertext c, outputs a KEM key $k \in \mathcal{K}$ or a rejection symbol $\perp \notin \mathcal{K}$. In this paper, we only focus on *implicit-rejection KEM*. Namely, we require that Decaps always outputs a KEM key in $\mathcal{K}$ for any input.

Some KEM schemes (such as lattice-based schemes) may not have perfect decryption correctness. To capture this, we define the correctness definition for KEM schemes in Definition 1.

Definition 1 (KEM Correctness). *We say* KEM *is* $(1-\delta)$*-correct if*

$$\Pr_{(c,k)\leftarrow\mathsf{Encaps}(\mathsf{pk})}[k = \mathsf{Decaps}(\mathsf{sk}, c)] \geq 1 - \delta,$$

where the probability is taken over $\mathsf{par} \leftarrow \mathsf{Setup}, (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}$ *and the randomness of* Encaps*. When using this definition in a security proof, we may use* δ *to bound the probability of decryption error.*

Definition 2 (OW-PCA security of KEM). KEM *is OW-PCA secure if, for all PPT* $\mathcal{A}$*, we have that the advantage* $\mathbf{Adv}^{\mathsf{OW\text{-}PCA}}_{\mathsf{KEM}}(\mathcal{A}) := \Pr\left[\mathsf{OW\text{-}PCA}^{\mathcal{A}}_{\mathsf{KEM}} \Rightarrow 1\right]$ *is negligible.*

Memory-tight KEM constructions. In Supp. Mat. E, we discuss how to construct an OW-PCA KEM from a standard IND-CPA PKE scheme with a memory-tight reduction. We also discuss how to lift OW-PCA security in the single-user, single-challenge setting to the multi-user, multi-challenge setting via a memory-tight but advantage-loose reduction.

2.2 Signature Schemes

We define multi-user strong unforgeability against adaptive corruptions and chosen-one-per-message attack (aka. MU^c-sEUF-1CMA) in Definition 3 that is needed for our memory-efficient AKE. We also define its weaker, single-user variant in Definition 3.

Definitions. A digital signature scheme with message space $\mathcal{M}$, signature space Σ, and randomness space $\mathcal{R}$ consists of four algorithms:

- Setup takes the security parameter λ as input and outputs a scheme parameter $\mathsf{par}_{\mathtt{sig}}$ for the signature scheme.
- KG takes $\mathsf{par}_{\mathtt{sig}}$ as input and outputs a key pair (pk, sk).
- Sign takes a secret key sk and a message $m \in \mathcal{M}$ as input and outputs a signature $\sigma \in \Sigma$.
- Vrfy takes a public key pk and a pair (m, σ) as input and outputs 0 or 1. We say σ is a valid signature of m under pk if $\mathsf{Vrfy}(\mathsf{pk}, m, \sigma) = 1$.

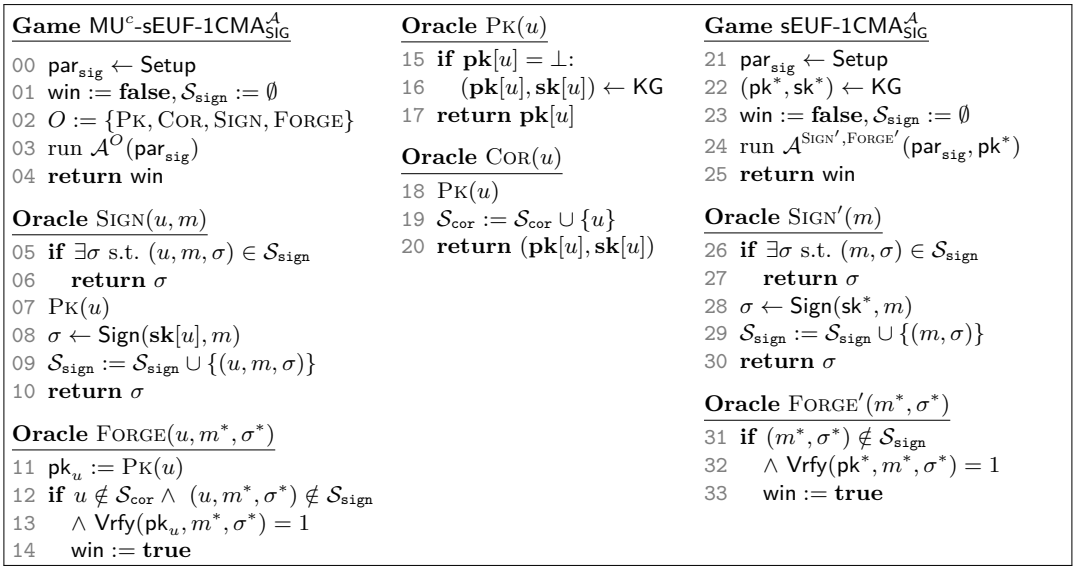

Fig. 2. In MU^c-sEUF-1CMA, $\mathcal{A}$ can obtain multiple public keys by querying Pk.

Since the security parameter and scheme parameter are fixed after setup, we ignore them and write $\mathsf{par}_{\mathtt{sig}} \leftarrow \mathsf{Setup}$ and $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}$. We specify the randomness in key generation and signing by writing $(\mathsf{pk}, \mathsf{sk}) := \mathsf{KG}(; r)$ and $\sigma := \mathsf{Sign}(\mathsf{sk}, m; r)$.

Let $\mathcal{U}$ be a set of user identifiers, and let $(\mathsf{pk}_u, \mathsf{sk}_u)$ be the key pair of user $u \in \mathcal{U}$. MU^c-sEUF-1CMA and sEUF-1CMA are defined in Definition 3. Discussion on how to define $\mathcal{U}$ can be found in Section 3.2 and [JK22, Section 2.6].

Definition 3 (Security for Signatures). *Let $\mathcal{A}$ be an adversary against* SIG. *We define two security games* MU^c-sEUF-1CMA *and* sEUF-1CMA *in Figure 2 and say* SIG *is* MU^c-sEUF-1CMA *(resp.* sEUF-1CMA*) secure, if, for all PPT adversaries $\mathcal{A}$, its advantage function defined as follows is negligible:*

$$\mathbf{Adv}^{\mathsf{MU}^c\text{-sEUF-1CMA}}_{\mathsf{SIG}}(\mathcal{A}) := \Pr\left[\mathsf{MU}^c\text{-sEUF-1CMA}^{\mathcal{A}}_{\mathsf{SIG}} \Rightarrow \mathbf{true}\right]$$
$$\mathbf{Adv}^{\mathsf{sEUF-1CMA}}_{\mathsf{SIG}}(\mathcal{A}) := \Pr\left[\mathsf{sEUF\text{-}1CMA}^{\mathcal{A}}_{\mathsf{SIG}} \Rightarrow \mathbf{true}\right]$$

Since the signature key pair will be used as a long-term key pair in our AKE construction, we also require that the probability that two different users have the same public key is negligible, as defined in Definition 4.

Definition 4 (Signature Public-Key Collisions). *For a positive integer N, we define the public-key collision probability of* SIG *as*

$$\eta_{pk}(N) := \Pr\left[\exists\, i, j \in [N],\ i \neq j : \mathsf{pk}_i = \mathsf{pk}_j\right],$$

where $\mathsf{par}_{\mathtt{sig}} \leftarrow \mathsf{Setup}$ *and* $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KG}$ *are generated independently for all $i \in [N]$ under the same parameter* $\mathsf{par}_{\mathtt{sig}}$.

Memory-tight Implication. In Supp. Mat. D, we show that MU^c-sEUF-1CMA can be implied by sEUF-1CMA. While the reduction is not TA-tight, the memory tightness is preserved (cf. Lemma 10).

2.3 Memory-tight Reductions

$\mathcal{F}$-Oracle Adversary. In this paper, we use the formalism of memory-tight reductions in [GGJT22,JK22]. All reduction adversaries in this paper have the form that can be divided into two parts: a part that simulates random functions, and a remaining part. The latter part cannot control the simulation of

random functions in the former part. More precisely, let $\mathcal{B}$ be a reduction algorithm. We can write $\mathcal{B}$ in the form shown below for some set(s) of functions $\mathcal{F}$ and algorithm $\mathcal{B}^*$:

Reduction $\mathcal{B}^O(inp)$
34 $f \leftarrow \mathcal{F}$
35 $out \leftarrow \mathcal{B}^{*O,f}(inp)$
36 **return** out

The time and memory complexity of $\mathcal{B}$ include the complexity from $\mathcal{B}^*$ and complexity from the sampling, storing, and computing $\mathcal{F}$. More specifically, the (full) running time and memory usage of $\mathcal{B}$ are written as

$$\mathbf{T}(\mathcal{B}) := \mathbf{T}(\mathcal{B}^*) + \mathbf{T}(f) \text{ and } \mathbf{Mem}(\mathcal{B}) := \mathbf{Mem}(\mathcal{B}^*) + \mathbf{Mem}(f).$$

As in [GGJT22], it is sufficient to use the reduced time complexity $\mathbf{T}^*(\mathcal{B})$ and reduced memory complexity $\mathbf{Mem}^*(\mathcal{B})$, which are defined as

$$\mathbf{T}^*(\mathcal{B}) := \mathbf{T}(\mathcal{B}^*) \text{ and } \mathbf{Mem}^*(\mathcal{B}) := \mathbf{Mem}(\mathcal{B}^*).$$

The argument in favor of using the reduced notions is as follows. Let $\mathcal{B}$ be an adversary against a computationally hard problem. If f is independent of the hard problem and is black-box accessed by $\mathcal{B}$, then it is very unlikely that the time and memory used to simulate f significantly help $\mathcal{B}$ to break the hard problem. To see this, a conservative way to measure how much the oracle f helps $\mathcal{B}$ is to replace f with a pseudorandom version of it. For example, when f is a random function, we can replace it with a pseudorandom function PRF so that we do not need memory-inefficient lazy sampling. Such an approach was also used in memory-tight reductions for signature, PKE, and KEM [ACFK17,Bha20,DGJL21a].

We recall the following lemma from [GGJT22] to account for such a replacement-via-pseudorandomness approach. For a function family F, let $\mathsf{F.K}$ be its key space and $\mathsf{F.F}$ its corresponding ideal function class. Let $\mathcal{B}_\mathsf{F}$ be obtained from $\mathcal{B}$ by sampling $s \leftarrow \mathsf{F.K}$ and answering queries to f using F_s. We write $\mathbf{Adv}^{\mathsf{pr}}_{\mathsf{F}}(\mathcal{B}')$ for the advantage of $\mathcal{B}'$ in distinguishing F_s, for a random key s, from a uniformly sampled $f \leftarrow \mathsf{F.F}$, with the same oracle interface in both cases.

Lemma 1 ([GGJT22], Lemma 2). *Let $\mathcal{B}$ be an $\mathcal{F}$-oracle adversary for a game $\mathbf{G}$. For any function family F with $\mathsf{F.F} = \mathcal{F}$, there exists a pseudorandomness adversary $\mathcal{B}'$ such that*

$$\begin{aligned}
&\Pr[\mathbf{G}(\mathcal{B}) \Rightarrow 1] \le \Pr[\mathbf{G}(\mathcal{B}_\mathsf{F}) \Rightarrow 1] + \mathbf{Adv}^{\mathsf{pr}}_{\mathsf{F}}(\mathcal{B}'), \quad \mathbf{Query}(\mathcal{B}') \le Q \\
&\mathbf{T}(\mathcal{B}') = \mathbf{T}^*(\mathcal{B}) + \mathbf{T}(\mathbf{G}(\mathcal{B})), \mathbf{Mem}(\mathcal{B}') = \mathbf{Mem}^*(\mathcal{B}) + \mathbf{Mem}(\mathbf{G}(\mathcal{B})), \\
&\mathbf{T}(\mathcal{B}_\mathsf{F}) \le \mathbf{T}^*(\mathcal{B}) + (Q+1) \cdot \mathbf{T}(\mathsf{F}), \mathbf{Mem}(\mathcal{B}_\mathsf{F}) \le \mathbf{Mem}^*(\mathcal{B}) + \mathbf{Mem}(\mathsf{F}).
\end{aligned}$$

Here Q is an upper bound on the number of queries $\mathcal{B}^*$ makes to f, including inverse queries when present. The terms $\mathbf{T}(\mathbf{G}(\mathcal{B}))$ and $\mathbf{Mem}(\mathbf{G}(\mathcal{B}))$ account for the game's own computation and storage when interacting with $\mathcal{B}$. $\mathbf{T}(\mathsf{F})$ upper-bounds the time for key generation and for each evaluation. $\mathbf{Mem}(\mathsf{F})$ includes the key and the workspace needed for key generation and evaluation.

After the replacement, the advantage of $\mathcal{B}$ is bounded by that of $\mathcal{B}_\mathsf{F}$ plus the pseudorandomness advantage $\mathbf{Adv}^{\mathsf{pr}}_{\mathsf{F}}(\mathcal{B}')$. For a PRF instantiation, the memory used to store the random values is replaced by the PRF key and evaluation workspace. Thus, the full memory usage of $\mathcal{B}_\mathsf{F}$ is $\mathbf{Mem}^*(\mathcal{B})$ plus this additional memory, and its full running time is $\mathbf{T}^*(\mathcal{B})$ plus the time for key generation and all PRF evaluations.

The distinguisher $\mathcal{B}'$ also simulates the security game $\mathbf{G}$, which may require large memory. We do not require this simulation to be memory-tight. As discussed in [GGJT22, Section 3], we can use a pseudorandomness bound that depends on the distinguisher's time, query counts, and memory. Since F can be chosen independently of the underlying problem, we can choose a sufficiently long key so that $\mathbf{Adv}^{\mathsf{pr}}_{\mathsf{F}}(\mathcal{B}')$ is negligible at these bounds (even when the memory is large). The key and evaluation workspace of such a long key can still be small compared with the memory used by the core adversary. In this way, we can replace the potentially large cost of simulating f without significantly changing the advantage bound. This observation motivates the definition of *reduced complexity*, which simplifies the statements of our theorems and helps us focus on the memory usage of the core adversary $\mathcal{B}^*$ without

fixing a particular pseudorandom instantiation. These arguments also apply to pseudorandom injective functions, with their security and evaluation costs covering the required inverse queries.

In this paper, we use reduced complexity to measure the running time and memory usage of reductions. In our proofs, $\mathcal{F}$ is normally a set of random functions or random injective functions with specific domains and codomains. One can instantiate random functions by PRFs, and random injections by suitable encryption schemes (or PRPs when the domain and codomain coincide), preserving the required inverse interface. The corresponding distinguishing terms and any statistical error of the instantiations are then added to the advantage bound. We refer readers to [GGJT22, Lemma 2 and Appendix A] and [JK22, Section 2.2] for these replacements.

For an $\mathcal{F}$-oracle adversary $\mathcal{B}$, if the context is clear, then we will ignore $\mathcal{F}$ and simply write it as adversary $\mathcal{B}$.

Based on the notations above, we say a reduction $\mathcal{B}$ (invoking some adversary $\mathcal{A}$) is *TA-tight* if $\mathbf{Adv}(\mathcal{A}) \leq \mathbf{Adv}(\mathcal{B}) + \mathsf{negl}(\lambda)$ and $\mathbf{T}(\mathcal{A}) \approx \mathbf{T}^*(\mathcal{B})$, and is *memory-tight* if $\mathbf{Mem}(\mathcal{A}) = \mathbf{Mem}^*(\mathcal{B}) + \mathrm{O}(\mathsf{poly}(\lambda))$, where λ is a security parameter. We say $\mathcal{B}$ is *TAM-tight* if it is both TA-tight and memory-tight.

Message Encoding Techniques. Let $\mathcal{X}$ and $\mathcal{Y}$ be two sets. If $|\mathcal{Y}| \geq |\mathcal{X}|$, then $\mathsf{InjF}(\mathcal{X}, \mathcal{Y})$ denotes the set of all injections from $\mathcal{X}$ to $\mathcal{Y}$. For $f \in \mathsf{InjF}(\mathcal{X}, \mathcal{Y})$, we write f^{-1} as its inverse. If $y \in \mathcal{Y}$ has no preimage, then $f^{-1}(y) = \perp$ where $\perp$ is a special symbol indicating "rejection" or "does not exist". If $y = (y_1, y_2) \in \mathcal{Y}_1 \times \mathcal{Y}_2$ is a tuple, then in this paper, we define that $y \neq \perp \iff y_1 \neq \perp$ or $y_2 \neq \perp$.

We use the message encoding technique proposed in [GGJT22,JK22] that helps reductions to hide state information in some randomly generated object. For example, let (a, b) be a random pair from $\mathcal{Y}^2$ and $x \in \mathcal{X}$ be some element that is coupled with (a, b), e.g., x is the index of (a, b). In the reduction, we can generate (a, b) by picking two random injections $f_1, f_2 : \mathcal{X} \to \mathcal{Y}$ and computing $a := f_1(x)$ and $b := f_2(x)$. Later, when the adversary gives a back to the reduction, then the reduction can retrieve x (and then retrieve b) by computing $x := f_1^{-1}(a)$. This technique allows the reduction to "forget" some information and thus is very useful for memory-tight reductions.

3 Our New Model for Authenticated Key Exchange

We propose a new security model for two-message AKE that captures the same security goals as in other models [JKRS21,PWZ23a,PWZ23b], but in a memory-efficient manner. Formally, we prove in Supp. Mat. C that a protocol secure in our model is also secure in a single-Test comparison model adapted from [PWZ23b].

The current AKE models usually store a large amount of information to simulate the security game with adversaries. As a consequence, it makes the security reduction unnecessarily memory-inefficient. Hence, we slightly revise the security model to facilitate the construction of memory-tight AKE protocols, while preserving the same security guarantees. Our syntax is defined (in Section 3.1) in a similar way as the previous work, and our new security definitions (in Section 3.2) are the major difference from the previous ones.

3.1 Syntax

A two-message AKE protocol $\mathsf{AKE} := (\mathsf{Setup}, \mathsf{KG}, \mathsf{Init}_\mathrm{I}, \mathsf{Der}_\mathrm{R}, \mathsf{Der}_\mathrm{I})$ consists of five algorithms as illustrated in Figure 3. The setup algorithm Setup takes the system security parameter 1^λ as input and outputs an AKE protocol parameter par. On input par, the key generation algorithm KG outputs an AKE long-term key pair $(\mathsf{pk}, \mathsf{sk})$. Since 1^λ and par are public and fixed after setting up, for simplicity, we ignore the inputs of Setup and KG and write $\mathsf{par} \leftarrow \mathsf{Setup}$ and $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}$.

The algorithms $\mathsf{Init}_\mathrm{I}, \mathsf{Der}_\mathrm{R}, \mathsf{Der}_\mathrm{I}$ also take the user identifiers (u, v), always ordered as initiator and responder. Let u, v be two users with long-term key pairs $(\mathsf{pk}_u, \mathsf{sk}_u), (\mathsf{pk}_v, \mathsf{sk}_v)$, respectively. To establish a shared session key, u, v perform the following steps:

- To initiate a session, u firstly runs the initialization algorithm Init_I which takes $(u, v, \mathsf{pk}_u, \mathsf{sk}_u, \mathsf{pk}_v)$ as inputs and outputs an initiator message M_u and session state st. Then u sends M_u to v. The state st is stored locally by u and will be used to compute a shared key later.

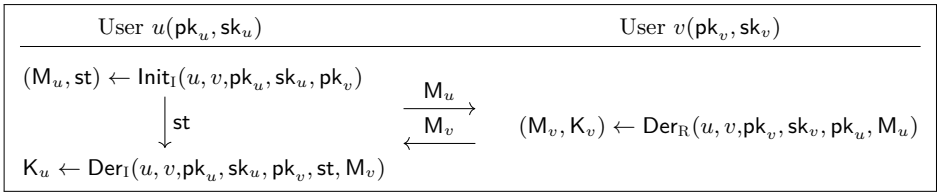

Fig. 3. Illustration for a two-message AKE protocol execution between users u and v.

- On receiving M_u from the user u, v runs the responder derivation algorithm Der_R. On input $(u, v, \mathsf{pk}_v, \mathsf{sk}_v, \mathsf{pk}_u, \mathsf{M}_u)$, this algorithm generates and returns a responder message M_v and a session key K_v. To continue the session, v sends M_v to u. K_v can be $\perp$ (rejection symbol) which means that the received message M_u is invalid.
- Finally, on receiving M_v, the user u restores the state st generated before and runs the initiator derivation algorithm $\mathsf{Der}_\mathrm{I}(u, v, \mathsf{pk}_u, \mathsf{sk}_u, \mathsf{pk}_v, \mathsf{st}, \mathsf{M}_v)$ to derive a session key K_u. Similarly, K_u can be $\perp$.

We define the correctness of AKE protocols in Definition 5 which states that an honest execution between users u and v as in Figure 3 should produce the same session key $\mathsf{K}_u = \mathsf{K}_v$.

Definition 5 (AKE Correctness). *Let* $\mathsf{AKE} := (\mathsf{Setup}, \mathsf{KG}, \mathsf{Init}_\mathrm{I}, \mathsf{Der}_\mathrm{R}, \mathsf{Der}_\mathrm{I})$ *be an AKE protocol and* u *and* v *be two users. We say* AKE *is* ρ*-correct if*

$$\Pr\left[\mathsf{K}_u = \mathsf{K}_v \neq \perp : \begin{array}{l} \mathsf{par} \leftarrow \mathsf{Setup} \\ (\mathsf{pk}_u, \mathsf{sk}_u) \leftarrow \mathsf{KG}, (\mathsf{pk}_v, \mathsf{sk}_v) \leftarrow \mathsf{KG} \\ (\mathsf{M}_u, \mathsf{st}) \leftarrow \mathsf{Init}_\mathrm{I}(u, v, \mathsf{pk}_u, \mathsf{sk}_u, \mathsf{pk}_v) \\ (\mathsf{M}_v, \mathsf{K}_v) \leftarrow \mathsf{Der}_\mathrm{R}(u, v, \mathsf{pk}_v, \mathsf{sk}_v, \mathsf{pk}_u, \mathsf{M}_u) \\ \mathsf{K}_u \leftarrow \mathsf{Der}_\mathrm{I}(u, v, \mathsf{pk}_u, \mathsf{sk}_u, \mathsf{pk}_v, \mathsf{st}, \mathsf{M}_v) \end{array}\right] \geq \rho,$$

where the probability is taken over the randomness of $\mathsf{Setup}, \mathsf{KG}, \mathsf{Init}_\mathrm{I}, \mathsf{Der}_\mathrm{R}$*, and* Der_I*.* ρ *is some function related to the security parameter* λ *(e.g.,* $\rho = 1 - \mathsf{negl}(\lambda)$*).*

3.2 Security for AKE

Our security game is in Figure 4. We explain how the useful variables are defined.

Naming users and generating pk. Our model considers a set of user identifiers, $\mathcal{U}$. Each user in our model has a unique identifier from $\mathcal{U}$. For simplicity, we use u to indicate a user with identifier $u \in \mathcal{U}$. The definition of $\mathcal{U}$ depends on the concrete real-world implementation of AKE protocols. Usually, we model $\mathcal{U}$ as a set of fixed-length bit-strings (namely, $\mathcal{U} = \{0,1\}^c$ for some fixed constant c). For the more general case $\mathcal{U} := \{0,1\}^*$, we will handle it as in [JK22, Section 2.6].

Let $\Pi = (\mathsf{Setup}, \mathsf{KG}, \mathsf{Init}_\mathrm{I}, \mathsf{Der}_\mathrm{R}, \mathsf{Der}_\mathrm{I})$ be an AKE protocol and $\mathsf{par} \leftarrow \mathsf{Setup}$ be the protocol parameter. The long-term key pair of a user u is denoted as $(\mathsf{pk}_u, \mathsf{sk}_u)$. In this model, the adversary $\mathcal{A}$ can access an oracle $\textsc{Pk}$ to learn the long-term public key of any user. When $\mathcal{A}$ queries $\textsc{Pk}(u \in \mathcal{U})$, the oracle returns pk_u. If $\mathcal{A}$ queries $\textsc{Pk}$ repeatedly on the same user identifier, then the same public key is returned. This method of modelling public key generation is proposed by [GGJT22,JK22]. It allows security reductions to simulate all public keys in a memory-tight manner and does not change the adversary's capabilities.

Modeling sessions. $\mathcal{A}$ has oracle access to $\textsc{Init}_\mathrm{I}, \textsc{Der}_\mathrm{R}$, and $\textsc{Der}_\mathrm{I}$. This is to capture the adversary's ability to control the transmission of all protocol messages and launch active attacks. Every query to $\textsc{Init}_\mathrm{I}$ and $\textsc{Der}_\mathrm{R}$ will produce a unique sequence number from a sequence number pool $\mathcal{SEQ}$. A query to $\textsc{Der}_\mathrm{I}$ will produce a unique sequence number which is determined by the query, namely, the same queries to $\textsc{Der}_\mathrm{I}$ will produce the same sequence numbers. More precisely,

- $\textsc{Init}_\mathrm{I}$ allows $\mathcal{A}$ to initiate a session between u (as an initiator and owner) and v (as a responder). When $\mathcal{A}$ queries $\textsc{Init}_\mathrm{I}(u, v)$, the oracle returns $(\mathsf{seq}, \mathsf{M}_u)$ where M_u is u's initiator protocol message (generated by running Init_I) and $\mathsf{seq} \in \mathcal{SEQ}$ is a uniformly random sequence number.
- $\textsc{Der}_\mathrm{R}$ allows $\mathcal{A}$ to send an arbitrary message M to v, pretending to be u. This creates a session owned by v. When $\mathcal{A}$ queries $\textsc{Der}_\mathrm{R}(u, v, \mathsf{M})$, the oracle runs Der_R to generate a responder message M_v and a session key K. If $\mathsf{K} = \perp$, then the oracle simply returns $\perp$; Otherwise, the oracle generates a unique sequence number seq uniformly at random and defines a session identifier sID for this session. Finally, $\textsc{Der}_\mathrm{R}$ returns sID, which contains M_v. $\mathcal{A}$ can use the identifier sID to test or reveal this session later.

Game ind-wFS-KCI$^{\mathcal{A}}_{\Pi}$

00 $b \leftarrow \{0,1\}$, $\mathsf{par} \leftarrow \mathsf{Setup}$
01 $O := (\textsc{Pk}, \textsc{Cor}, \textsc{Init}_{\mathrm{I}}, \textsc{Rev}, \textsc{Test}, \textsc{Der}_{\mathrm{I}}, \textsc{Der}_{\mathrm{R}})$
02 $b' \leftarrow \mathcal{A}^{O}(\mathsf{par})$
03 **for** $\mathsf{sID}^* \in \mathcal{S}_{\mathtt{test}}$:
04 **if** $\mathsf{Fresh}(\mathsf{sID}^*) = 0$
05 **return** $b' \leftarrow \{0,1\}$
06 **if** $\mathsf{Valid}(\mathsf{sID}^*) = 0$
07 **return** $b' \leftarrow \{0,1\}$
08 **return** $[\![b' = b]\!]$

Oracle $\textsc{Init}_{\mathrm{I}}(u, v)$

09 $\textsc{Pk}(u)$, $(\mathsf{pk}_u, \mathsf{sk}_u) := \mathcal{L}_{\mathtt{key}}[u]$
10 $(\mathsf{M}_u, \mathsf{st}) \leftarrow \mathsf{Init}_{\mathrm{I}}(u, v, \mathsf{pk}_u, \mathsf{sk}_u, \mathsf{pk}_v)$
11 $\mathsf{seq} \leftarrow \mathcal{SEQ} \backslash \mathcal{S}_{\mathtt{seq}}$, $\mathcal{S}_{\mathtt{seq}} := \mathcal{S}_{\mathtt{seq}} \cup \{\mathsf{seq}\}$
12 $\mathcal{L}_{\mathtt{st}}[\mathsf{seq}] := (u, v, \mathsf{M}_u, \mathsf{st})$
13 $\mathcal{S}_{\mathtt{msg}} := \mathcal{S}_{\mathtt{msg}} \cup \{(u, \mathsf{M}_u)\}$
14 **return** $(\mathsf{seq}, \mathsf{M}_u)$

Oracle $\textsc{Der}_{\mathrm{R}}(u, v, \mathsf{M}_u)$

15 $\textsc{Pk}(v)$, $\mathsf{pk}_u \leftarrow \textsc{Pk}(u)$
16 $(\mathsf{pk}_v, \mathsf{sk}_v) := \mathcal{L}_{\mathtt{key}}[v]$
17 $(\mathsf{M}_v, \mathsf{K}) \leftarrow \mathsf{Der}_{\mathrm{R}}(u, v, \mathsf{pk}_v, \mathsf{sk}_v, \mathsf{pk}_u, \mathsf{M}_u)$
18 **if** $\mathsf{K} = \bot$: **return** $\bot$
19 **if** $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$
20 $\mathcal{S}_{\mathtt{msg}} := \mathcal{S}_{\mathtt{msg}} \cup \{(u, v, \mathsf{M}_u, \mathsf{M}_v)\}$
21 $\mathsf{seq} \leftarrow \mathcal{SEQ} \backslash \mathcal{S}_{\mathtt{seq}}$
22 $\mathsf{sID} := (\mathsf{seq}, u, v, \mathsf{M}_u, \mathsf{M}_v)$
23 $\mathcal{S}_{\mathtt{seq}} := \mathcal{S}_{\mathtt{seq}} \cup \{\mathsf{seq}\}$
24 $\mathcal{S}_{\mathtt{sid}} := \mathcal{S}_{\mathtt{sid}} \cup \{\mathsf{sID}\}$
25 $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}] := \mathsf{K}$
26 **return** sID

Oracle $\textsc{Der}_{\mathrm{I}}(u, v, \mathsf{seq}_1, \mathsf{M}_v)$

27 **if** $\mathsf{seq}_1 \notin \mathcal{S}_{\mathtt{seq}}$: **return** $\bot$
28 $\textsc{Pk}(u), \mathsf{pk}_v \leftarrow \textsc{Pk}(v)$
29 $(\mathsf{pk}_u, \mathsf{sk}_u) := \mathcal{L}_{\mathtt{key}}[u]$
30 $(u', v', \mathsf{M}_u, \mathsf{st}) := \mathcal{L}_{\mathtt{st}}[\mathsf{seq}_1]$
31 **if** $(u', v') \neq (u, v) \vee \mathsf{st} = \bot$
32 **return** $\bot$
33 $\mathsf{K} := \mathsf{Der}_{\mathrm{I}}(u, v, \mathsf{pk}_u, \mathsf{sk}_u, \mathsf{pk}_v, \mathsf{st}, \mathsf{M}_v)$
34 **if** $\mathsf{K} = \bot$: **return** $\bot$
35 **if** $\mathcal{L}_{\mathtt{seq}}[\mathsf{seq}_1, u, v, \mathsf{M}_u, \mathsf{M}_v] = \bot$
36 $\mathsf{seq}_2 \leftarrow \mathcal{SEQ} \backslash \mathcal{S}_{\mathtt{seq}}$
37 $\mathcal{S}_{\mathtt{seq}} := \mathcal{S}_{\mathtt{seq}} \cup \{\mathsf{seq}_2\}$
38 $\mathcal{L}_{\mathtt{seq}}[\mathsf{seq}_1, u, v, \mathsf{M}_u, \mathsf{M}_v] := \mathsf{seq}_2$
39 $\mathsf{seq}_2 := \mathcal{L}_{\mathtt{seq}}[\mathsf{seq}_1, u, v, \mathsf{M}_u, \mathsf{M}_v]$
40 $\mathsf{sID} := (\mathsf{seq}_1, \mathsf{seq}_2, u, v, \mathsf{M}_u, \mathsf{M}_v)$
41 $\mathcal{S}_{\mathtt{sid}} := \mathcal{S}_{\mathtt{sid}} \cup \{\mathsf{sID}\}$
42 $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}] := \mathsf{K}$
43 **return** sID

Oracle $\textsc{Pk}(u \in \mathcal{U})$

44 $(\mathsf{pk}_u, \mathsf{sk}_u) \leftarrow \mathsf{GetKey}(u)$
45 **return** pk_u

Oracle $\textsc{Cor}(u)$

46 $(\mathsf{pk}_u, \mathsf{sk}_u) := \mathsf{GetKey}(u)$
47 $\mathcal{S}_{\mathtt{cor}} := \mathcal{S}_{\mathtt{cor}} \cup \{u\}$
48 **return** sk_u

Oracle $\textsc{Rev}(\mathsf{sID})$

49 **if** $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}] = \bot$: **return** $\bot$
50 $\mathcal{S}_{\mathtt{rev}} := \mathcal{S}_{\mathtt{rev}} \cup \{\mathsf{sID}\}$
51 **return** $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}]$

Alg $\mathsf{GetKey}(u)$

52 **if** $\mathcal{L}_{\mathtt{key}}[u] = \bot$
53 $\mathcal{L}_{\mathtt{key}}[u] := (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}$
54 **return** $\mathcal{L}_{\mathtt{key}}[u]$

Oracle $\textsc{Test}(\mathsf{sID})$

55 **if** $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}] = \bot$: **return** $\bot$
56 $\mathcal{S}_{\mathtt{test}} := \mathcal{S}_{\mathtt{test}} \cup \{\mathsf{sID}\}$
57 $(*, u, v, \mathsf{M}_u, \mathsf{M}_v) := \mathsf{sID}$
58 **if** $\mathcal{L}_{\mathtt{test}}[u, v, \mathsf{M}_u, \mathsf{M}_v] = \bot$:
59 $\mathcal{L}_{\mathtt{test}}[u, v, \mathsf{M}_u, \mathsf{M}_v] \leftarrow \mathcal{SK}$
60 $\mathsf{SK}_0 := \mathcal{L}_{\mathtt{SK}}[\mathsf{sID}]$
61 $\mathsf{SK}_1 := \mathcal{L}_{\mathtt{test}}[u, v, \mathsf{M}_u, \mathsf{M}_v]$
62 **return** SK_b

Alg $\mathsf{ComputeMS}(\mathsf{sID})$

63 $(*, u, v, \mathsf{M}_u, \mathsf{M}_v) := \mathsf{sID}$
64 **for** $\mathsf{sID}' \in \mathcal{S}_{\mathtt{sid}} \backslash \{\mathsf{sID}\}$
65 $(*, u', v', \mathsf{M}'_u, \mathsf{M}'_v) := \mathsf{sID}'$
66 **if** $(u, v, \mathsf{M}_u, \mathsf{M}_v) = (u', v', \mathsf{M}'_u, \mathsf{M}'_v)$
67 $\mathfrak{M}(\mathsf{sID}) := \mathfrak{M}(\mathsf{sID}) \cup \{\mathsf{sID}'\}$
68 **return** $\mathfrak{M}(\mathsf{sID})$

Alg $\mathsf{Fresh}(\mathsf{sID})$

69 $\mathfrak{M}(\mathsf{sID}) := \mathsf{ComputeMS}(\mathsf{sID})$
70 **if** $|\mathfrak{M}(\mathsf{sID})| > 1$: **return** 1
71 **if** $\mathsf{sID} \in \mathcal{S}_{\mathtt{rev}}$: **return** 0
72 **for** $\mathsf{sID}' \in \mathfrak{M}(\mathsf{sID})$:
73 **if** $\mathsf{sID}' \in \mathcal{S}_{\mathtt{rev}}$: **return** 0
74 **return** 1

Alg $\mathsf{Valid}(\mathsf{sID})$

75 **if** $\mathsf{sID} = (\mathsf{seq}, u, v, \mathsf{M}_u, \mathsf{M}_v)$ // Responder session
76 **if** $u \notin \mathcal{S}_{\mathtt{cor}} \vee (u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$
77 **return** 1
78 **if** $\mathsf{sID} = (\mathsf{seq}_1, \mathsf{seq}_2, u, v, \mathsf{M}_u, \mathsf{M}_v)$ // Initiator session
79 **if** $v \notin \mathcal{S}_{\mathtt{cor}} \vee (u, v, \mathsf{M}_u, \mathsf{M}_v) \in \mathcal{S}_{\mathtt{msg}}$
80 **return** 1
81 **return** 0

Fig. 4. Game ind-wFS-KCI. For simplicity, we use procedure GetKey to generate and store a user's long-term key pair, and use procedures ComputeMS, Fresh, and Valid to check if a session is fresh or is valid (cf. Definition 6). The adversary cannot access these procedures.

– DER_I allows $\mathcal{A}$ to complete an initiator session by sending a message. When $\mathcal{A}$ queries $\text{DER}_\text{I}(u, v, \mathsf{seq}_1, \mathsf{M})$, the oracle restores session state information st related to seq_1. If such seq_1 is not generated from previous INIT_I queries, then the oracle simply returns $\perp$; Otherwise, it runs Der_I to generate a session key K. Similar to DER_R, if K is $\perp$, then the oracle returns $\perp$. Otherwise, the oracle generates a sequence number seq_2 related to this query, defines a session identifier sID for this session, and returns sID.

For the above oracles, we use the following lists and sets to keep track of some necessary internal information (which an adversary cannot directly access). All lists and sets are initially set as empty:

– Lists $\mathcal{L}_{\mathtt{key}}$ and $\mathcal{L}_{\mathtt{SK}}$ are used to store user long-term keys and session keys, respectively.
– List $\mathcal{L}_{\mathtt{st}}$ is used to store session states of initiator sessions and the identifiers of intended responders.
– List $\mathcal{L}_{\mathtt{seq}}$ is used to store the sequence numbers generated by Der_I. By using this list, the model makes sure that the same query to Der_I will produce the same session final sequence number seq_2 (cf. lines 35 to 38).
– Sets $\mathcal{S}_{\mathtt{seq}}$ and $\mathcal{S}_{\mathtt{msg}}$ are used to store session sequence numbers and protocol messages (with the identifier of the user that generates this message) generated by oracles, respectively.

  More specifically, $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$ means that $\mathcal{A}$ has queried $\text{INIT}_\text{I}(u, v)$ before and the query produced the protocol message M_u. If $\mathcal{A}$ later sends this (u, M_u) to some v (via querying DER_R), then we record $(u, v, \mathsf{M}_u, \mathsf{M}_v) \in \mathcal{S}_{\mathtt{msg}}$ to help us determine if there exists a matching session, where M_v is the responding message produced by the party v that responds to M_u.
– Set $\mathcal{S}_{\mathtt{sid}}$ stores all session identifiers. This set is used when we compute the number of matching sessions. We have $\mathsf{sID} \in \mathcal{S}_{\mathtt{sid}} \iff \mathcal{L}_{\mathtt{SK}}[\mathsf{sID}] \neq \perp$.

Defining session identifiers. For an initiator session owned by a party u, if its intended peer is v, sent message is M_u, and received message is M_v, then its identifier is of the format $\mathsf{sID} := (\mathsf{seq}_1, \mathsf{seq}_2, u, v, \mathsf{M}_u, \mathsf{M}_v)$, where seq_1 and seq_2 are the initial and final sequence numbers generated by INIT_I and DER_I. Similarly, the identifier of a responder session is defined as $\mathsf{sID} := (\mathsf{seq}, u, v, \mathsf{M}_u, \mathsf{M}_v)$, where seq is the sequence number generated by DER_R.

Let sID and sID' be an initiator and responder session, respectively. We say sID and sID' are matching sessions, if their user identifiers of initiator and responder and the protocol messages sent and received are the same. Namely, $\mathsf{sID} = (\mathsf{seq}_1, \mathsf{seq}_2, u, v, \mathsf{M}_u, \mathsf{M}_v)$ matches $\mathsf{sID}' = (\mathsf{seq}', u', v', \mathsf{M}'_u, \mathsf{M}'_v)$ if $(u, v, \mathsf{M}_u, \mathsf{M}_v) = (u', v', \mathsf{M}'_u, \mathsf{M}'_v)$.

Modelling attacks and security. We define *freshness* and *validity* to exclude all trivial attacks. In this model, we use auxiliary procedures Fresh and Valid (cf. Figure 4) to check whether a session is fresh and valid, respectively. For a full understanding of the attacks defined in our model, please refer to Supp. Mat. B. In short, our model captures the same attacks defined in [JKRS21,PWZ23a,PWZ23b], including replay attacks, man-in-the-middle (MiM) attacks, key-compromise impersonation (KCI) attacks, and weak forward-secrecy (wFS). A difference is that we do not consider partially matching sessions in this model (cf. Remark 2).

Definition 6 (Freshness and Validity). *A complete session* sID *is fresh if either (1)* $\mathsf{sID} \notin \mathcal{S}_{\mathtt{rev}}$ *(the session is not revealed) and for any* $\mathsf{sID}' \in \mathfrak{M}(\mathsf{sID})$*, we have* $\mathsf{sID}' \notin \mathcal{S}_{\mathtt{rev}}$ *(the matching session is also not revealed); or (2)* $|\mathfrak{M}(\mathsf{sID})| > 1$ *(there is more than one matching session).*

Furthermore, a session is valid if at least one of the following conditions holds true: (1) If the session is a responder session, then either the initiator in the session is uncorrupted or the initiator message of this session is generated from INIT_I*. (2) If this session is an initiator session, then either the responder in this session is uncorrupted or this session has a matching session.*

Following the previous indistinguishability (IND)-based AKE model [JKRS21,PWZ23a], our model is the multi-Test model with a single random bit b. Let sID be an arbitrary established session, $\text{TEST}(\mathsf{sID})$ returns the real session key of sID if $b = 0$ and returns an independently and uniformly random key if $b = 1$. To deal with the case that $\mathcal{A}$ issues repeating queries to Test or tests both a session and its matching session, we use a list $\mathcal{L}_{\mathtt{test}}$ to record the returning keys of Test (cf. lines 57 to 59). The formal security definition of our AKE model is given below.

Definition 7 (Key Indistinguishability of AKE). *Let* $\Pi := (\mathsf{Setup}, \mathsf{KG}, \mathsf{Init}_\text{I}, \mathsf{Der}_\text{R}, \mathsf{Der}_\text{I})$ *be an AKE protocol and consider the game* $\mathsf{ind\text{-}wFS\text{-}KCI}$ *defined in Figure 4. We say* AKE *is secure if, for all PPT*

Alg Setup

00 $\mathsf{par}_{\mathtt{kem}} \leftarrow \mathsf{KEM.Setup}$
01 $\mathsf{par}_{\mathtt{sig}} \leftarrow \mathsf{SIG.Setup}$
02 **return** $\mathsf{par} := (\mathsf{par}_{\mathtt{kem}}, \mathsf{par}_{\mathtt{sig}})$

Alg $\mathsf{KG}(\mathsf{par}_{\mathtt{kem}}, \mathsf{par}_{\mathtt{sig}})$

03 $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{SIG.KG}(\mathsf{par}_{\mathtt{sig}})$
04 **return** $(\mathsf{pk}, \mathsf{sk})$

Alg $\mathsf{Init}_{\mathrm{I}}(u, v, \mathsf{pk}_u, \mathsf{sk}_u, \mathsf{pk}_v)$

05 $\alpha \leftarrow \{0,1\}^{\ell}$, $(epk, esk) \leftarrow \mathsf{KEM.KG}$
06 $\sigma_1 \leftarrow \mathsf{Sign}(\mathsf{sk}_u, (u, \alpha, epk))$
07 $\mathsf{M}_u := (\alpha, epk, \sigma_1)$, $\mathsf{st} := (esk, \mathsf{M}_u)$
08 **return** $(\mathsf{M}_u, \mathsf{st})$

Alg $\mathsf{Der}_{\mathrm{R}}(u, v, \mathsf{pk}_v, \mathsf{sk}_v, \mathsf{pk}_u, (\alpha, epk, \sigma_1))$

09 **if** $\mathsf{Vrfy}(\mathsf{pk}_u, (u, \alpha, epk), \sigma_1) = 0$
10 **return** $\perp$
11 $\beta \leftarrow \{0,1\}^{\ell}$, $(c, k) \leftarrow \mathsf{Encaps}(epk)$
12 $\sigma_2 \leftarrow \mathsf{Sign}(\mathsf{sk}_v, (u, v, \alpha, \beta, epk, c, \sigma_1))$
13 $\mathsf{K} := \mathsf{H}(\mathsf{pk}_u, \mathsf{pk}_v, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k)$
14 $\mathsf{M}_v := (\beta, c, \sigma_2)$
15 **return** $(\mathsf{M}_v, \mathsf{K})$

Alg $\mathsf{Der}_{\mathrm{I}}(u, v, \mathsf{pk}_u, \mathsf{sk}_u, \mathsf{pk}_v, \mathsf{st}, (\beta, c, \sigma_2))$

16 $(esk, \alpha, epk, \sigma_1) := \mathsf{st}$
17 **if** $\mathsf{Vrfy}(\mathsf{pk}_v, (u, v, \alpha, \beta, epk, c, \sigma_1), \sigma_2) = 0$
18 **return** $\perp$
19 $k := \mathsf{Decaps}(esk, c)$
20 **if** $k = \perp$: **return** $\perp$
21 $\mathsf{K} := \mathsf{H}(\mathsf{pk}_u, \mathsf{pk}_v, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k)$
22 **return** K

Fig. 5. Our AKE protocol aSigKEM.

adversaries $\mathcal{A}$, the following advantage is negligible in the security parameter λ:

$$\mathbf{Adv}^{\mathsf{ind\text{-}wFSKCI}}_{\Pi}(\mathcal{A}) := \left| \Pr\left[\mathsf{ind\text{-}wFS\text{-}KCI}^{\mathcal{A}}_{\Pi} \Rightarrow 1 \right] - 1/2 \right|$$

4 Our Generic AKE from KEM and Signature

Our new AKE protocol is an abstraction of the well-known signed Diffie-Hellman key exchange protocol. Motivated by the augmented-random-string technique in [JK22] and the signed Diffie-Hellman protocol discussed in the technical overview in Section 1, our protocol replaces the DH part with CCA-secure KEM and includes a random nonce in each move, and thus we call it augmented Signed KEM (aSigKEM) protocol.

Construction. Let $\mathsf{KEM} := (\mathsf{Setup}, \mathsf{KG}, \mathsf{Encaps}, \mathsf{Decaps})$ be a KEM scheme and $\mathsf{SIG} := (\mathsf{Setup}, \mathsf{KG}, \mathsf{Sign}, \mathsf{Vrfy})$ be a signature scheme. Let $\{0,1\}^{\ell}$ be a nonce space, $\mathcal{SK}$ be a session key space, and $\mathsf{H} : \{0,1\}^* \to \mathcal{SK}$ be a hash function. Our protocol aSigKEM is defined in Figure 5. We assume that all tuples used as signature messages or random-function inputs are encoded unambiguously.

Theorem 1. *For any adversary $\mathcal{A}$ against* aSigKEM *as in our model* ind-wFS-KCI *with user identifier set $\mathcal{U}$ and session sequence number set $\mathcal{SEQ}$ (cf. Figure 4), if $\mathcal{A}$ issues $Q_{\mathrm{PK}}, Q_{\mathrm{REV}}, Q_{\mathrm{COR}}, Q_{\mathrm{TEST}}, Q_{\mathsf{H}}$, and $Q_{\mathtt{send}}$ queries to oracles* PK *(including direct and indirect queries),* REV, COR, TEST, H, *and* $(\mathrm{INIT_I}, \mathrm{DER_I}, \mathrm{DER_R})$, *$|\mathcal{SEQ}| \geq |\mathcal{U}|^2 \cdot N_{\mathtt{ses}}$, and $2^{\ell} \geq 2|\mathcal{U}|^2 \cdot N_{\mathtt{ses}} \cdot (N_{\mathtt{ses}} + 1)$, then there exist adversaries $\mathcal{B}_{\mathsf{SIG}}$ and $\mathcal{B}_{\mathsf{KEM}}$ such that*

$$\begin{aligned}
\mathbf{Adv}^{\mathsf{ind\text{-}wFSKCI}}_{\mathtt{aSigKEM}}(\mathcal{A}) &\leq \mathbf{Adv}^{\mathsf{OW\text{-}PCA}}_{\mathsf{KEM}}(\mathcal{B}_{\mathsf{KEM}}) + \mathbf{Adv}^{\mathsf{MU^c\text{-}sEUF\text{-}1CMA}}_{\mathsf{SIG}}(\mathcal{B}_{\mathsf{SIG}}) \\
&\quad + \frac{(Q_{\mathsf{H}} + N_{\mathtt{ses}})^2 + Q_{\mathsf{H}} \cdot N_{\mathtt{ses}}}{|\mathcal{K}|} + N^2_{\mathtt{ses}} \delta_{\mathsf{KEM}} + \eta_{pk}(N_{\mathtt{user}}) \\
&\quad + \frac{N_{\mathtt{ses}}(5N_{\mathtt{ses}} + 3Q_{\mathsf{H}})(|\mathcal{U}| + 1)}{2^{\ell}} + \frac{4N^2_{\mathtt{ses}} + Q_{\mathrm{REV}} + Q_{\mathrm{TEST}}}{|\mathcal{SEQ}|} \\
\mathbf{Mem}^*(\mathcal{B}_{\mathsf{KEM}}) &\approx \mathbf{Mem}^*(\mathcal{B}_{\mathsf{SIG}}) = \mathbf{Mem}(\mathcal{A}) + \mathrm{O}(\log |\mathcal{U}|) + \mathrm{O}(\ell) \\
&\quad + \mathrm{O}(\log |\mathcal{SEQ}|) + \mathrm{O}(\log N_{\mathtt{ses}}) + \mathbf{Mem}(\mathtt{aSigKEM}), \\
\mathbf{T}^*(\mathcal{B}_{\mathsf{KEM}}) &\approx \mathbf{T}^*(\mathcal{B}_{\mathsf{SIG}}) = \mathbf{T}(\mathcal{A}) + \mathrm{O}(Q_{\mathtt{query}} + Q_{\mathsf{H}}) \cdot \mathbf{T}(\mathtt{aSigKEM}),
\end{aligned}$$

where $\mathcal{SEQ}$ is the space of available sequence numbers, ℓ is the length of nonces of aSigKEM, *δ_{KEM} is the decryption error bound of* KEM, *$N_{\mathtt{user}}$ is the number of (non-repeated) users participating in the* ind-wFS-KCI *game, $\eta_{pk}(N_{\mathtt{user}})$ is the collision probability of $N_{\mathtt{user}}$ public keys generated by* SIG.KG *(cf. Definition 4), and $N_{\mathtt{ses}} := Q_{\mathtt{send}}$ (the upper bound on the number of sessions established by $\mathcal{A}$). Let $Q_{\mathtt{query}}$ be the total number of queries made directly by $\mathcal{A}$ to* $\{\mathrm{PK}, \mathrm{COR}, \mathrm{REV}, \mathrm{TEST}, \mathrm{INIT_I}, \mathrm{DER_I}, \mathrm{DER_R}\}$,

counting repeated and rejected queries. It excludes queries to H *and internal oracle calls.* $\mathbf{T}(\mathtt{aSigKEM})$ *and* $\mathbf{Mem}(\mathtt{aSigKEM})$ *are the required running time and memory usage of running a complete local execution of* $\mathtt{aSigKEM}$*, respectively.* $\mathbf{T}(\mathtt{aSigKEM})$ *also upper-bounds long-term key generation.*

For a large-scale application, we can let the set of user identifiers be $\mathcal{U} = \{0,1\}^{32}$ and the set of session sequence numbers be $\mathcal{SEQ} = \{0,1\}^{256}$. If we consider a setting with $(N_{\mathtt{user}}, N_{\mathtt{ses}}) \approx (2^{32}, 2^{48})$, then we can choose $\ell = 320$ (which means that the nonces α and β are 40-byte strings).

Pseudorandom Instantiations. Let $\mathsf{F}_{\mathsf{KEM}}$ and $\mathsf{F}_{\mathsf{SIG}}$ be the pseudorandom verions of the auxiliary random functions and injections in $\mathcal{B}_{\mathsf{KEM}}$ and $\mathcal{B}_{\mathsf{SIG}}$, respectively, and let $\widehat{\mathcal{B}_{\mathsf{KEM}}}$ and $\widehat{\mathcal{B}_{\mathsf{SIG}}}$ denote the resulting reductions instantiated with $\mathsf{F}_{\mathsf{KEM}}$ and $\mathsf{F}_{\mathsf{SIG}}$. For notational convenience (and also without loss of generality), each family collects the independently keyed implementations, and the keys of $\mathsf{F}_{\mathsf{KEM}}$ and $\mathsf{F}_{\mathsf{SIG}}$ are the tuple of the keys of these auxiliary functions and injections, and each query specifies the function to evaluate or invert.

By Lemma 1, there exist distinguishers $\mathcal{B}'_{\mathsf{KEM}}$ and $\mathcal{B}'_{\mathsf{SIG}}$ such that

$$\mathbf{Adv}^{\mathsf{ind\text{-}wFSKCI}}_{\mathtt{aSigKEM}}(\mathcal{A}) \le \mathbf{Adv}^{\mathsf{OW\text{-}PCA}}_{\mathsf{KEM}}(\widehat{\mathcal{B}_{\mathsf{KEM}}}) + \mathbf{Adv}^{\mathsf{MU}^c\text{-}\mathsf{sEUF\text{-}1CMA}}_{\mathsf{SIG}}(\widehat{\mathcal{B}_{\mathsf{SIG}}}) + \epsilon_{\mathrm{stat}} + \epsilon_{\mathrm{pr}},$$
$$\epsilon_{\mathrm{pr}} \coloneqq \mathbf{Adv}^{\mathsf{pr}}_{\mathsf{F}_{\mathsf{KEM}}}(\mathcal{B}'_{\mathsf{KEM}}) + \mathbf{Adv}^{\mathsf{pr}}_{\mathsf{F}_{\mathsf{SIG}}}(\mathcal{B}'_{\mathsf{SIG}}).$$

Here ϵ_{stat} is the sum of the remaining terms in the advantage bound of Theorem 1. The two terms in ϵ_{pr} measure how well the pseudorandom implementations can be distinguished from the original random functions and injections in the KEM and signature reductions, respectively. The full time and memory bounds follow from Lemma 1, using $\mathsf{F}_{\mathsf{KEM}}$ and $\mathsf{F}_{\mathsf{SIG}}$ for the respective reductions.

Interpreting Bounds in Theorem 1. Our bounds are TAM-tight wrt the security of the KEM and signature schemes. This significantly improves other AKE protocols on the prior works [GJ18,JKRS21,PQR22,PWZ23a,FSXY12,HKSU20,PWZ23b] in terms of memory-tightness, as we remove the linear memory-loss factors $\mathrm{O}(N_{\mathtt{user}})$, $\mathrm{O}(N_{\mathtt{ses}})$, and $\mathrm{O}(Q_{\mathsf{RO}})$ as in Equations (1) and (2).

Proof (Theorem 1). Let $\mathcal{A}$ be an adversary against $\mathtt{aSigKEM}$ in the $\mathsf{ind\text{-}wFS\text{-}KCI}$ game. Our proof consists of a sequence of games as in Figures 6 to 8.

Game $\mathbf{G}_0$. $\mathbf{G}_0$ is shown in Figure 6. Compared to the original security game $\mathsf{ind\text{-}wFS\text{-}KCI}$, the RO outputs of H in $\mathbf{G}_0$ are generated by the random function $f_{\mathtt{H}}$ (cf. line 40), and the random keys of Test are generated by the random function $f_{\mathtt{rSK}}$ (cf. line 18). Moreover, we remove the case that $k = \perp$ in $\mathrm{DER_I}$ or $\mathrm{DER_R}$, and assume that different users have different public keys. Since the distributions of H and Test remain unchanged and we assume KEM has implicit rejection in Theorem 1, by Definition 4, we have

$$\left|\Pr\left[\mathsf{ind\text{-}wFS\text{-}KCI}^{\mathcal{A}}_{\mathtt{aSigKEM}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_0^{\mathcal{A}} \Rightarrow 1\right]\right| \le \eta_{pk}(N_{\mathtt{user}})$$

Game $\mathbf{G}_1$ (Use RFs to derive randomness). In this game, we generate randomness for KEM and SIG using random functions instead of generating them inside the algorithms. For the signature scheme SIG, $f_{\mathtt{skg}}$ (cf. line 22 in Figure 6), and $f_{\mathtt{sI}}$ and $f_{\mathtt{sR}}$ (cf. lines 31 and 54 in Figure 6) are used for signature key generation and the signing process. For the KEM scheme KEM, we first set up a global counter $\mathsf{cnt} \in [N_{\mathtt{ses}}]$ which is initialized to 0 and incremented by 1 every time it is used (cf. lines 24 and 43 in Figure 6). Since there are at most $N_{\mathtt{ses}}$ queries to $\mathrm{Init_I}$ and $\mathrm{DER_R}$, the counter will not overflow. We then use this counter to compute the ephemeral KEM key pairs from $\mathsf{KEM.KG}$ and encapsulated ciphertexts from $\mathsf{KEM.Encaps}$ using random functions $f_{\mathtt{ekg}}$ and $f_{\mathtt{enc}}$, respectively (cf. line 26 and lines 44 to 45 in Figure 6).

The bound on distinguishing $\mathbf{G}_0$ and $\mathbf{G}_1$ is summarized in Lemma 2. As the proof is straightforward, we defer the details to Supp. Mat. F.1.

All inputs to random functions for KEM are unique for each query, so no collision happens and this change is perfectly indistinguishable. For the signature scheme SIG, the inputs to random functions are unique per session, assuming that the partial message (i.e., (α, epk) and (β, c)) to be signed is unique for each session, which holds with high probability if the nonce space $\{0,1\}^{\ell}$ is large enough. Thus, we only lose a birthday bound, as stated in Lemma 2.

Game $\mathbf{G}_0$-$\mathbf{G}_3$

00 $\mathsf{cnt} \coloneqq 0$ // $\mathbf{G}_{[1\text{-}3]}$
01 $f_{\mathtt{skg}} \leftarrow \mathsf{Func}(\mathcal{U}, \mathcal{R}_{\mathtt{k}})$ // $\mathbf{G}_{[1\text{-}3]}$
02 $f_{\mathtt{sI}} \leftarrow \mathsf{Func}(\mathcal{U}\times\{0,1\}^{\ell} \times \mathcal{PK}, \mathcal{R}_{\mathtt{s}})$ // $\mathbf{G}_{[1\text{-}3]}$
03 $f_{\mathtt{sR}} \leftarrow \mathsf{Func}(\mathcal{U}^2\times(\{0,1\}^{\ell})^2 \times \mathcal{PK} \times\mathcal{C}\times\Sigma, \mathcal{R}_{\mathtt{s}})$ // $\mathbf{G}_{[1\text{-}3]}$
04 $f_{\mathtt{ekg}} \leftarrow \mathsf{Func}([N_{\mathtt{ses}}], \mathcal{R}_{\mathtt{ekg}})$ // $\mathbf{G}_{[1\text{-}3]}$
05 $f_{\mathtt{enc}} \leftarrow \mathsf{Func}(\mathcal{PK} \times [N_{\mathtt{ses}}], \mathcal{R}_{\mathtt{enc}})$ // $\mathbf{G}_{[1\text{-}3]}$
06 $f_{\mathtt{seq}} \leftarrow \mathsf{Func}(\{0,1\}^*, \mathcal{SEQ})$ // $\mathbf{G}_{[2\text{-}3]}$
07 $f'_{\mathtt{seq}} \leftarrow \mathsf{Func}(\mathcal{U}^2 \times [N_{\mathtt{ses}}], \mathcal{SEQ})$ // $\mathbf{G}_2$
08 $h_{\mathtt{seq}} \leftarrow \mathsf{InjF}(\mathcal{U}^2 \times [N_{\mathtt{ses}}], \mathcal{SEQ})$ // $\mathbf{G}_3$
09 $h_{\mathtt{nce}} \leftarrow \mathsf{InjF}(\{0,1\} \times \mathcal{U}^2 \times(N_{\mathtt{ses}}] \times [N_{\mathtt{ses}}], \{0,1\}^{\ell})$ // $\mathbf{G}_3$
10 $f_{\mathtt{H}}, f_{\mathtt{rSK}} \leftarrow \mathsf{Func}(\{0,1\}^*, \mathcal{SK})$
11 $b \leftarrow \{0,1\}$, $\mathsf{par} \leftarrow \mathsf{Setup}$
12 $O_1 \coloneqq (\textsc{Init}_{\mathrm{I}}, \textsc{Der}_{\mathrm{I}}, \textsc{Der}_{\mathrm{R}})$
13 $O_2 \coloneqq (\textsc{Pk}, \textsc{Cor}, \textsc{Rev}, \textsc{Test})$
14 $b' \leftarrow \mathcal{A}^{O_1,O_2,\mathsf{H}}(\mathsf{par})$
15 **for** $\mathsf{sID}^* \in \mathcal{S}_{\mathtt{test}}$:
16 **if** $\mathsf{Fresh}(\mathsf{sID}^*) = 0 \vee \mathsf{Valid}(\mathsf{sID}^*) = 0$
17 **return** $b' \leftarrow \{0,1\}$
18 **return** $[\![b' = b]\!]$

Alg $\mathsf{GetKey}(u)$

19 **if** $\mathcal{L}_{\mathtt{key}}[u] = \bot$ // $\mathbf{G}_0$
20 $\mathcal{L}_{\mathtt{key}}[u] \coloneqq (\mathsf{pk}_u, \mathsf{sk}_u) \leftarrow \mathsf{SIG.KG}$ // $\mathbf{G}_0$
21 **return** $\mathcal{L}_{\mathtt{key}}[u]$ // $\mathbf{G}_0$
22 $(\mathsf{pk}_u, \mathsf{sk}_u) \coloneqq \mathsf{SIG.KG}(f_{\mathtt{skg}}(u))$ // $\mathbf{G}_{[1\text{-}3]}$
23 **return** $(\mathsf{pk}_u, \mathsf{sk}_u)$ // $\mathbf{G}_{[1\text{-}3]}$

Oracle $\textsc{Init}_{\mathrm{I}}(u, v)$

24 $\mathsf{cnt} \coloneqq \mathsf{cnt} + 1$ // $\mathbf{G}_{[1\text{-}3]}$
25 $(epk, esk) \leftarrow \mathsf{KEM.KG}$ // $\mathbf{G}_0$
26 $(epk, esk) \coloneqq \mathsf{KEM.KG}(; f_{\mathtt{ekg}}(\mathsf{cnt}))$ // $\mathbf{G}_{[1\text{-}3]}$
27 $\alpha \leftarrow \{0,1\}^{\ell}$ // $\mathbf{G}_{[0\text{-}2]}$
28 $\alpha \coloneqq h_{\mathtt{nce}}(0, u, v, 0, \mathsf{cnt})$ // $\mathbf{G}_3$
29 $\sigma_1 \leftarrow \mathsf{Sign}(\mathsf{sk}_u, (u,\alpha, epk))$ // $\mathbf{G}_0$
30 $r_{\mathtt{sign}} \coloneqq f_{\mathtt{sI}}(u,\alpha, epk)$ // $\mathbf{G}_{[1\text{-}3]}$
31 $\sigma_1 \coloneqq \mathsf{Sign}(\mathsf{sk}_u, (u,\alpha, epk); r_{\mathtt{sign}})$ // $\mathbf{G}_{[1\text{-}3]}$
32 $\mathsf{seq}_1 \leftarrow \mathcal{SEQ}\backslash\mathcal{S}_{\mathtt{seq}}$ // $\mathbf{G}_0$-$\mathbf{G}_1$
33 $\mathcal{S}_{\mathtt{seq}} \coloneqq \mathcal{S}_{\mathtt{seq}} \cup \{\mathsf{seq}_1\}$ // $\mathbf{G}_0$-$\mathbf{G}_1$
34 $\mathsf{seq}_1 \coloneqq f'_{\mathtt{seq}}(u, v, \mathsf{cnt})$ // $\mathbf{G}_2$
35 $\mathsf{seq}_1 \coloneqq h_{\mathtt{seq}}(u, v, \mathsf{cnt})$ // $\mathbf{G}_3$
36 $\mathsf{M}_u \coloneqq (\alpha, (epk, \sigma_1))$
37 $\mathsf{st} \coloneqq (esk, \mathsf{M}_u)$, $\mathcal{L}_{\mathtt{st}}[\mathsf{seq}_1] \coloneqq (u, v, \mathsf{st})$
38 $\mathcal{S}_{\mathtt{msg}} \coloneqq \mathcal{S}_{\mathtt{msg}} \cup \{(u, \mathsf{M}_u)\}$
39 **return** $(\mathsf{seq}_1, \mathsf{M}_u)$

Oracle $\mathsf{H}(\mathsf{pk}_1, \mathsf{pk}_2, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k)$

40 **return** $f_{\mathtt{H}}(\mathsf{pk}_1, \mathsf{pk}_2, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k)$

Oracle $\textsc{Der}_{\mathrm{R}}(u, v, \mathsf{M}_u = (\alpha, epk, \sigma_1))$

41 **if** $\mathsf{Vrfy}(\mathsf{pk}_u, (u,\alpha, epk), \sigma_1) = 0$: **return** $\bot$
42 $(c, k) \leftarrow \mathsf{KEM.Encaps}(epk)$ // $\mathbf{G}_0$
43 $\mathsf{cnt} \coloneqq \mathsf{cnt} + 1$ // $\mathbf{G}_{[1\text{-}3]}$
44 $y \coloneqq f_{\mathtt{enc}}(epk, \mathsf{cnt})$ // $\mathbf{G}_{[1\text{-}3]}$
45 $(c, k) \coloneqq \mathsf{KEM.Encaps}(epk; y)$ // $\mathbf{G}_{[1\text{-}3]}$
46 $\beta \leftarrow \{0,1\}^{\ell}$ // $\mathbf{G}_{[0\text{-}2]}$
47 **if** $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$ // $\mathbf{G}_3$
48 $(*, \mathsf{cnt}_1) \coloneqq h^{-1}_{\mathtt{nce}}(\alpha)$ // $\mathbf{G}_3$
49 $\beta \coloneqq h_{\mathtt{nce}}(1, u, v, \mathsf{cnt}_1, \mathsf{cnt})$ // $\mathbf{G}_3$
50 **else** // $\mathbf{G}_3$
51 $\beta \coloneqq h_{\mathtt{nce}}(1, u, v, 0, \mathsf{cnt})$ // $\mathbf{G}_3$
52 $\sigma_2 \leftarrow \mathsf{Sign}(\mathsf{sk}_v, (u, v,\alpha, \beta, epk, c, \sigma_1))$ // $\mathbf{G}_0$
53 $r_{\mathtt{sign}} \coloneqq f_{\mathtt{sR}}(u, v,\alpha, \beta, epk, c, \sigma_1)$ // $\mathbf{G}_{[1\text{-}3]}$
54 $\sigma_2 \coloneqq \mathsf{Sign}(\mathsf{sk}_v, (u, v, \alpha, \beta, epk, c, \sigma_1); r_{\mathtt{sign}})$ // $\mathbf{G}_{[1\text{-}3]}$
55 $\mathsf{K} \coloneqq \mathsf{H}(\mathsf{pk}_u, \mathsf{pk}_v, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k)$
56 $\mathsf{M}_v \coloneqq (\beta, c, \sigma_2)$
57 **if** $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$
58 $\mathcal{S}_{\mathtt{msg}} \coloneqq \mathcal{S}_{\mathtt{msg}} \cup \{(u, v, \mathsf{M}_u, \mathsf{M}_v)\}$
59 $\mathsf{seq} \leftarrow \mathcal{SEQ}\backslash\mathcal{S}_{\mathtt{seq}}$, $\mathcal{S}_{\mathtt{seq}} \coloneqq \mathcal{S}_{\mathtt{seq}} \cup \{\mathsf{seq}\}$ // $\mathbf{G}_0$-$\mathbf{G}_1$
60 $\mathsf{seq} \coloneqq f_{\mathtt{seq}}(u, v, \mathsf{M}_u, \mathsf{M}_v)$ // $\mathbf{G}_{[2\text{-}3]}$
61 $\mathsf{sID} \coloneqq (\mathsf{seq}, u, v, \mathsf{M}_u, \mathsf{M}_v)$
62 $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}] \coloneqq \mathsf{K}$, $\mathcal{S}_{\mathtt{sid}} \coloneqq \mathcal{S}_{\mathtt{sid}} \cup \{\mathsf{sID}\}$
63 **return** sID

Oracle $\textsc{Der}_{\mathrm{I}}(u, v, \mathsf{seq}_1, \mathsf{M}_v = (\beta, c, \sigma_2))$

64 $(u', v', \mathsf{st}) \coloneqq \mathcal{L}_{\mathtt{st}}[\mathsf{seq}_1]$
65 **if** $(u', v') \neq (u, v) \vee \mathsf{st} = \bot$: **return** $\bot$
66 $(esk, \mathsf{M}_u = (\alpha, epk, \sigma_1)) \coloneqq \mathsf{st}$
67 **if** $\mathsf{Vrfy}(\mathsf{pk}_v, (u, v,\alpha, \beta, epk, c, \sigma_1), \sigma_2) = 0$
68 **return** $\bot$
69 $k \coloneqq \mathsf{Decaps}(esk, c)$
70 **if** $k = \bot$: **return** $\bot$
71 $\mathsf{K} \coloneqq \mathsf{H}(\mathsf{pk}_u, \mathsf{pk}_v, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k)$
72 **if** $\mathcal{L}_{\mathtt{seq}}[\mathsf{seq}_1, u, v, \mathsf{M}_u, \mathsf{M}_v] = \bot$ // $\mathbf{G}_{[0\text{-}1]}$
73 $\mathsf{seq}_2 \leftarrow \mathcal{SEQ}\backslash\mathcal{S}_{\mathtt{seq}}$ // $\mathbf{G}_{[0\text{-}1]}$
74 $\mathcal{S}_{\mathtt{seq}} \coloneqq \mathcal{S}_{\mathtt{seq}} \cup \{\mathsf{seq}_2\}$ // $\mathbf{G}_{[0\text{-}1]}$
75 $\mathcal{L}_{\mathtt{seq}}[\mathsf{seq}_1, u, v, \mathsf{M}_u, \mathsf{M}_v] \coloneqq \mathsf{seq}_2$ // $\mathbf{G}_{[0\text{-}1]}$
76 $\mathsf{seq}_2 \coloneqq \mathcal{L}_{\mathtt{seq}}[\mathsf{seq}_1, u, v, \mathsf{M}_u, \mathsf{M}_v]$ // $\mathbf{G}_{[0\text{-}1]}$
77 $\mathsf{seq}_2 \coloneqq f_{\mathtt{seq}}(\mathsf{seq}_1, u, v, \mathsf{M}_u, \mathsf{M}_v)$ // $\mathbf{G}_{[2\text{-}3]}$
78 $\mathsf{sID} \coloneqq (\mathsf{seq}_1, \mathsf{seq}_2, u, v, \mathsf{M}_u, \mathsf{M}_v)$
79 $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}] \coloneqq \mathsf{K}$, $\mathcal{S}_{\mathtt{sid}} \coloneqq \mathcal{S}_{\mathtt{sid}} \cup \{\mathsf{sID}\}$
80 **return** sID

Fig. 6. Games $\mathbf{G}_{[0\text{-}3]}$ for proving Theorem 1. For simplicity, we assume that in $\textsc{Init}_{\mathrm{I}}, \textsc{Der}_{\mathrm{R}}, \textsc{Der}_{\mathrm{I}}$, for any party U, we have $(\mathsf{pk}_{\mathsf{U}}, \mathsf{sk}_{\mathsf{U}}) = \mathsf{GetKey}(\mathsf{U})$. All omitted oracles and algorithms are the same as those in Figure 4.

Lemma 2. *With the notations and assumptions as stated in Theorem 1, we have*

$$\left|\Pr\left[\mathbf{G}_0^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_1^{\mathcal{A}} \Rightarrow 1\right]\right| \leq N_{\mathsf{ses}}^2 \cdot 2^{-\ell}$$

Game $\mathbf{G}_2$ (Random sequence number and sID). In this game, we change the generation of sequence numbers seq_1, seq, and seq_2 in $\textsc{Init}_{\mathrm{I}}$, $\textsc{Der}_{\mathrm{I}}$, and $\textsc{Der}_{\mathrm{R}}$ (cf. lines 34, 60 and 77 in Figure 6), respectively. Instead of sampling them uniformly at random from $\mathcal{SEQ}\backslash\mathcal{S}_{\mathsf{seq}}$, we generate them using random functions f'_{seq} and f_{seq}.

Due to space limitations, we only give a high-level proof sketch here and summarize it in Lemma 3. The detailed proof is presented in Supp. Mat. F.2. The change is essentially to replace the random sampling without replacement of sequence numbers with random functions (random sampling with replacement), which can be bounded by a standard birthday bound. To further ensure that different sessions will not

collide on the inputs to random functions, we use the unique session information (e.g., the counter cnt and the nonces α and β) as part of the inputs to random functions. By a union bound over all sessions, the probability that a collision happens is negligible as long as the sequence number space $\mathcal{SEQ}$ is large enough and the nonce space $\{0,1\}^\ell$ is large enough.

Lemma 3. *With the notations and assumptions as stated in Theorem 1, we have*

$$\left|\Pr\left[\mathbf{G}_1^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_2^{\mathcal{A}} \Rightarrow 1\right]\right| \leq \frac{N_{\mathsf{ses}}^2}{|\mathcal{SEQ}|} + \frac{N_{\mathsf{ses}}^2}{2^\ell}$$

Game $\mathbf{G}_3$ (Invertible nonces and initiator sequence number). We use a random injective function $h_{\mathtt{nce}}$ to generate the random nonces α's and β's and use $h_{\mathtt{seq}}$ to generate sequence numbers of $\textsc{Init}_{\mathrm{I}}$ (cf. lines 28 and 35 and lines 47 to 51 in Figure 6). That is, we replace uniform sampling of α and β and the random function $f_{\mathtt{seq}}$ by random injective functions $h_{\mathtt{nce}}$ and $h_{\mathtt{seq}}$, respectively, and the distinguishing advantage is bounded by their birthday collision probabilities. Since there are at most N_{ses} queries to $h_{\mathtt{nce}}$ and $h_{\mathtt{seq}}$ (via querying $\textsc{Init}_{\mathrm{I}}$ and $\textsc{Der}_{\mathrm{R}}$), we have

$$\left|\Pr\left[\mathbf{G}_2^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_3^{\mathcal{A}} \Rightarrow 1\right]\right| \leq N_{\mathsf{ses}}^2 \cdot 2^{-\ell-1} + N_{\mathsf{ses}}^2 \cdot (2|\mathcal{SEQ}|)^{-1}$$

In $h_{\mathtt{nce}}$, the function input includes a bit $\in \{0,1\}$ to distinguish nonces from $\textsc{Init}_{\mathrm{I}}$ and nonces from $\textsc{Der}_{\mathrm{R}}$. This ensures that the generated α's and β's have different preimages . Moreover, in $\textsc{Der}_{\mathrm{R}}$, we distinguish the case when the input message $\mathsf{M}(= (\alpha, epk, \sigma_1))$ is from the intended initiator u. If it is, we compute $\alpha = h_{\mathtt{nce}}(1, u, v, \mathsf{cnt}_1, \mathsf{cnt})$, where cnt_1 is the counter value of α. Otherwise, we compute $\alpha = h_{\mathtt{nce}}(0, u, v, 0, \mathsf{cnt})$. Note that $h_{\mathtt{nce}}^{-1}(\alpha)$ exists since α is generated by $h_{\mathtt{nce}}$ in $\textsc{Init}_{\mathrm{I}}$.

By the changes made in game sequence $\mathbf{G}_0$-$\mathbf{G}_3$, the game simulator now can forget the lists $\mathcal{L}_{\mathtt{seq}}$ and $\mathcal{L}_{\mathtt{key}}$ and the set $\mathcal{S}_{\mathtt{seq}}$. Moreover, by using injective random functions, we hide state information in all nonces and all initiator first sequence numbers (i.e., seq_1's) in the game. This allows the game simulator to restore such state information without using $\mathcal{L}_{\mathtt{st}}$, $\mathcal{L}_{\mathtt{SK}}$, and $\mathcal{S}_{\mathtt{msg}}$ in the future games. For readability, we present the codes of $\mathbf{G}_3$-$\mathbf{G}_6$ in Figure 7.

Game $\mathbf{G}_4$ (Simulate initiator states without recording). In this game, we restore initiator states $(esk, \alpha, epk, \sigma)$ by inverting the state information hidden in seq instead of using the list $\mathcal{L}_{\mathtt{st}}$ (cf. lines 62 to 66 in Figure 7).

More specifically, when $\mathcal{A}$ queries $\textsc{Der}_{\mathrm{I}}(u, v, \mathsf{seq}_1, \mathsf{M}_v)$, we invert seq_1 and get (u', v', cnt'), and check if $u = u'$, $v = v'$, and $\mathsf{cnt} < \mathsf{cnt}'$ (where cnt is the global counter in the game). If so, then we re-compute esk and (α, epk, σ_1) (by following lines 27 to 30). We summarize the analysis of $\mathbf{G}_4$ as a lemma and defer its proof to Supp. Mat. F.3.

Lemma 4. *With the notations and assumptions in Theorem 1, we have*

$$\left|\Pr\left[\mathbf{G}_3^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_4^{\mathcal{A}} \Rightarrow 1\right]\right| \leq \frac{2N_{\mathsf{ses}}^2}{|\mathcal{SEQ}|}$$

Game $\mathbf{G}_5$ (Check if an sID exists without recording). We define an algorithm $\mathsf{CheckSid}$ (in Figure 7) to check if a session identifier sID exists. This game is a preparatory step for forgetting the list $\mathcal{L}_{\mathtt{SK}}$. Concretely, when $\mathcal{A}$ queries $\textsc{Test}(\mathsf{sID})$ or $\textsc{Rev}(\mathsf{sID})$, we use $\mathsf{CheckSid}$ to check if the session identifier sID exists, namely, to check if sID is generated by $\textsc{Der}_{\mathrm{I}}$ or $\textsc{Der}_{\mathrm{R}}$ and has completed.

To distinguish $\mathbf{G}_5$ from $\mathbf{G}_4$, $\mathcal{A}$ needs to forge a session identifier sID that does not exist (namely, sID is not output from $\textsc{Der}_{\mathrm{R}}$ or $\textsc{Der}_{\mathrm{I}}$), but makes $\mathsf{CheckSid}$ return 1. And to forge such sID, $\mathcal{A}$ has to compute the correct sequence numbers seq (for responder sessions) or seq_2 (for initiator sessions) of sID.

For both responder sessions and initiator sessions, the final sequence number (e.g., seq or seq_2) is generated from the random function $f_{\mathtt{seq}}$ (cf. lines 60 and 77 in Figure 7) with inputs that capture the view of the session. Forging such a session identifier sID is equivalent to finding a fresh and valid input-output pair of a random function (i.e., random sampling with replacement). The only way for $\mathcal{A}$ to verify if it finds such a pair is to query $\textsc{Test}$ and $\textsc{Rev}$. Hence, if $\mathcal{A}$ queries $\textsc{Test}$ and $\textsc{Rev}$ at most $Q_{\textsc{Rev}} + Q_{\textsc{Test}}$ times, then we have

$$\left|\Pr\left[\mathbf{G}_4^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_5^{\mathcal{A}} \Rightarrow 1\right]\right| \leq \frac{Q_{\textsc{Rev}} + Q_{\textsc{Test}}}{|\mathcal{SEQ}|}$$

Game $\mathbf{G}_3$-$\mathbf{G}_6$

00 $\mathsf{cnt} \coloneqq 0$, $b \leftarrow \{0,1\}$, $\mathsf{par} \leftarrow \mathsf{Setup}$
01 Initialize $f_{\mathtt{seq}}, f_{\mathtt{H}}, f_{\mathtt{skg}}, f_{\mathtt{sI}}, f_{\mathtt{sR}}, f_{\mathtt{ekg}}$, $f_{\mathtt{enc}}, f_{\mathtt{rSK}}, h_{\mathtt{nce}}$, and $h_{\mathtt{seq}}$ as in $\mathbf{G}_3$
02 $f_{\mathtt{k}} \leftarrow \mathsf{Func}(\{0,1\}^*, \mathcal{K})$ $/\!/\mathbf{G}_6$
03 $O_1 \coloneqq (\textsc{Init}_{\mathrm{I}}, \textsc{Der}_{\mathrm{I}}, \textsc{Der}_{\mathrm{R}})$
04 $O_2 \coloneqq (\textsc{Pk}, \textsc{Cor}, \textsc{Rev}, \textsc{Test})$
05 $b' \leftarrow \mathcal{A}^{O_1,O_2,\mathsf{H}}(\mathsf{par})$
06 **for** $\mathsf{sID}^* \in \mathcal{S}_{\mathtt{test}}$:
07 **if** $\mathsf{Fresh}(\mathsf{sID}^*) = 0$
08 $\vee\ \mathsf{Valid}(\mathsf{sID}^*) = 0$
09 **return** $b' \leftarrow \{0,1\}$
10 **return** $[\![b' = b]\!]$

Oracle $\textsc{Test}(\mathsf{sID})$

11 **if** $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}] = \bot$: **return** $\bot$ $/\!/\mathbf{G}_{[3\text{-}4]}$
12 **if** $\mathsf{CheckSid}(\mathsf{sID}) = 0$ $/\!/\mathbf{G}_{[5\text{-}6]}$
13 **return** $\bot$ $/\!/\mathbf{G}_{[5\text{-}6]}$
14 $\mathcal{S}_{\mathtt{test}} \coloneqq \mathcal{S}_{\mathtt{test}} \cup \{\mathsf{sID}\}$
15 **parse** $(*, u, v, \mathsf{M}_u, \mathsf{M}_v) \coloneqq \mathsf{sID}$
16 $\mathsf{SK}_0 \coloneqq \mathcal{L}_{\mathtt{SK}}[\mathsf{sID}]$ $/\!/\mathbf{G}_{[3\text{-}5]}$
17 $\mathsf{SK}_0 = \mathsf{ComputeSK}(\mathsf{sID})$ $/\!/\mathbf{G}_6$
18 $\mathsf{SK}_1 \coloneqq f_{\mathtt{rSK}}(u, v, \mathsf{M}_u, \mathsf{M}_v)$
19 **return** SK_b

Oracle $\textsc{Rev}(\mathsf{sID})$

20 **if** $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}] = \bot$: **return** $\bot$ $/\!/\mathbf{G}_{[3\text{-}4]}$
21 **if** $\mathsf{CheckSid}(\mathsf{sID}) = 0$ $/\!/\mathbf{G}_{[5\text{-}6]}$
22 **return** $\bot$ $/\!/\mathbf{G}_{[5\text{-}6]}$
23 $\mathcal{S}_{\mathtt{rev}} \coloneqq \mathcal{S}_{\mathtt{rev}} \cup \{\mathsf{sID}\}$
24 **return** $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}]$ $/\!/\mathbf{G}_{[3\text{-}5]}$
25 **return** $\mathsf{ComputeSK}(\mathsf{sID})$ $/\!/\mathbf{G}_6$

Oracle $\textsc{Init}_{\mathrm{I}}(u, v)$

26 $\mathsf{cnt} \coloneqq \mathsf{cnt} + 1$
27 $\mathsf{seq}_1 \coloneqq h_{\mathtt{seq}}(u, v, \mathsf{cnt})$
28 $\alpha \coloneqq h_{\mathtt{nce}}(0, u, v, 0, \mathsf{cnt})$
29 $(epk, esk) \coloneqq \mathsf{KEM.KG}(; f_{\mathtt{ekg}}(\mathsf{cnt}))$
30 $\sigma \coloneqq \mathsf{Sign}(\mathsf{sk}_u, (u,\alpha, epk); f_{\mathtt{sI}}(u,\alpha, epk))$
31 $\mathsf{M}_u \coloneqq (\alpha, epk, \sigma)$
32 $\mathcal{L}_{\mathtt{st}}[\mathsf{seq}_1] \coloneqq (u, v, (esk, \mathsf{M}_u))$ $/\!/\mathbf{G}_3$
33 $\mathcal{S}_{\mathtt{msg}} \coloneqq \mathcal{S}_{\mathtt{msg}} \cup \{(u, \mathsf{M}_u)\}$
34 **return** $(\mathsf{seq}_1, \mathsf{M}_u)$

Alg $\mathsf{CheckSid}(\mathsf{sID})$

35 //Case: Responder session
36 **if** $\mathsf{sID} = (\mathsf{seq}, u, v, \mathsf{M}_u, \mathsf{M}_v)$
37 **return** $[\![\mathsf{seq} = f_{\mathtt{seq}}(u, v, \mathsf{M}_u, \mathsf{M}_v)]\!]$
38 //Case: Initiator session
39 **if** $\mathsf{sID} = (\mathsf{seq}_1, \mathsf{seq}_2, u, v, \mathsf{M}_u, \mathsf{M}_v)$
40 **return**
$[\![\mathsf{seq}_2 = f_{\mathtt{seq}}(\mathsf{seq}_1, u, v, \mathsf{M}_u, \mathsf{M}_v)]\!]$
41 **return** 0

Oracle $\textsc{Der}_{\mathrm{R}}(u, v, \mathsf{M}_u = (\alpha, epk, \sigma_1))$

42 **if** $\mathsf{Vrfy}(\mathsf{pk}_u, (u,\alpha, epk), \sigma_1) = 0$: **return** $\bot$
43 $\mathsf{cnt} \coloneqq \mathsf{cnt} + 1$
44 **if** $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$
45 $(*, \mathsf{cnt}_1) \coloneqq h^{-1}_{\mathtt{nce}}(\alpha)$, $\beta \coloneqq h_{\mathtt{nce}}(1, u, v, \mathsf{cnt}_1, \mathsf{cnt})$
46 **else** $\beta \coloneqq h_{\mathtt{nce}}(1, u, v, 0, \mathsf{cnt})$
47 $(c, k) \coloneqq \mathsf{KEM.Encaps}(epk; f_{\mathtt{enc}}(epk, \mathsf{cnt}))$
48 $r_{\mathtt{sign}} \coloneqq f_{\mathtt{sR}}(u, v,\alpha, \beta, epk, c, \sigma_1)$
49 $\sigma_2 \coloneqq \mathsf{Sign}(\mathsf{sk}_v, (u, v,\alpha, \beta, epk, c, \sigma_1); r_{\mathtt{sign}})$
50 $\mathsf{M}_v \coloneqq (\beta, c, \sigma_2)$
51 **if** $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$: $\mathcal{S}_{\mathtt{msg}} \coloneqq \mathcal{S}_{\mathtt{msg}} \cup \{(u, v, \mathsf{M}_u, \mathsf{M}_v)\}$
52 $\mathsf{seq} \coloneqq f_{\mathtt{seq}}(u, v, \mathsf{M}_u, \mathsf{M}_v)$
53 $\mathsf{sID} \coloneqq (\mathsf{seq}, u, v, \mathsf{M}_u, \mathsf{M}_v)$
54 $\mathsf{K} \coloneqq \mathsf{H}(\mathsf{pk}_u, \mathsf{pk}_v, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k)$ $/\!/\mathbf{G}_{[3\text{-}5]}$
55 $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}] \coloneqq \mathsf{K}$ $/\!/\mathbf{G}_{[3\text{-}5]}$
56 $\mathsf{K} \coloneqq \mathsf{ComputeSK}(\mathsf{sID})$ $/\!/\mathbf{G}_6$
57 $\mathcal{S}_{\mathtt{sid}} \coloneqq \mathcal{S}_{\mathtt{sid}} \cup \{\mathsf{sID}\}$ $/\!/\mathbf{G}_{[3\text{-}4]}$
58 **return** sID

Oracle $\textsc{Der}_{\mathrm{I}}(u, v, \mathsf{seq}_1, \mathsf{M}_v = (\beta, c, \sigma_2))$

59 **if** $\mathcal{L}_{\mathtt{st}}[\mathsf{seq}_1] = \bot$: **return** $\bot$ $/\!/\mathbf{G}_3$
60 $(u', v', (esk,\alpha, epk, \sigma_1)) \coloneqq \mathcal{L}_{\mathtt{st}}[\mathsf{seq}_1]$ $/\!/\mathbf{G}_3$
61 **if** $(u', v') \neq (u, v)$: **return** $\bot$ $/\!/\mathbf{G}_3$
62 $(u', v', \mathsf{cnt}') \coloneqq h^{-1}_{\mathtt{seq}}(\mathsf{seq}_1)$ $/\!/\mathbf{G}_{[4\text{-}6]}$
63 **if** $(u', v') \neq (u, v) \vee \mathsf{cnt} < \mathsf{cnt}'$: **return** $\bot$ $/\!/\mathbf{G}_{[4\text{-}6]}$
64 $\alpha \coloneqq h_{\mathtt{nce}}(0, u, v, 0, \mathsf{cnt}')$ $/\!/\mathbf{G}_{[4\text{-}6]}$
65 $x \coloneqq f_{\mathtt{ekg}}(\mathsf{cnt}')$, $(epk, esk) \coloneqq \mathsf{KEM.KG}(; x)$ $/\!/\mathbf{G}_{[4\text{-}6]}$
66 $\sigma_1 \coloneqq \mathsf{Sign}(\mathsf{sk}_u, (u,\alpha, epk); f_{\mathtt{sI}}(u,\alpha, epk))$ $/\!/\mathbf{G}_{[4\text{-}6]}$
67 **if** $\mathsf{Vrfy}(\mathsf{pk}_v, (u, v,\alpha, \beta, epk, c, \sigma_1), \sigma_2) = 0$: **return** $\bot$
68 $k \coloneqq \mathsf{Decaps}(esk, c)$ $/\!/\mathbf{G}_{[3\text{-}5]}$
69 $\mathsf{K} \coloneqq \mathsf{H}(\mathsf{pk}_u, \mathsf{pk}_v, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k)$ $/\!/\mathbf{G}_{[3\text{-}5]}$
70 $\mathsf{M}_u \coloneqq (\alpha, epk, \sigma_1)$
71 $\mathsf{seq}_2 \coloneqq f_{\mathtt{seq}}(\mathsf{seq}_1, u, v, \mathsf{M}_u, \mathsf{M}_v)$
72 $\mathsf{sID} \coloneqq (\mathsf{seq}_1, \mathsf{seq}_2, u, v, \mathsf{M}_u, \mathsf{M}_v)$
73 $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}] \coloneqq \mathsf{K}$ $/\!/\mathbf{G}_{[3\text{-}5]}$
74 $\mathsf{K} \coloneqq \mathsf{ComputeSK}(\mathsf{sID})$ $/\!/\mathbf{G}_6$
75 $\mathcal{S}_{\mathtt{sid}} \coloneqq \mathcal{S}_{\mathtt{sid}} \cup \{\mathsf{sID}\}$ $/\!/\mathbf{G}_{[3\text{-}4]}$
76 **return** sID

Alg $\mathsf{ComputeSK}(\mathsf{sID})$

77 $(*, u, v, (\alpha, epk, \sigma_1), (\beta, c, \sigma_2)) \coloneqq \mathsf{sID}$
78 **return** $f_{\mathtt{H}}(\mathsf{pk}_u, \mathsf{pk}_v, \alpha, \beta, epk, c, \sigma_1, \sigma_2, f_{\mathtt{k}}(epk, c))$

Oracle $\mathsf{H}(\mathsf{pk}_1, \mathsf{pk}_2, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k)$

79 $k^* \coloneqq k$, $d \coloneqq 0$
80 $(b_1, u_1, v_1, \mathsf{cnt}'_1, \mathsf{cnt}_1) \coloneqq h^{-1}_{\mathtt{nce}}(\alpha)$ $/\!/\mathbf{G}_6$
81 $(b_2, u_2, v_2, \mathsf{cnt}'_2, \mathsf{cnt}_2) \coloneqq h^{-1}_{\mathtt{nce}}(\beta)$ $/\!/\mathbf{G}_6$
82 **if** $(b_1, u_1, v_1, \mathsf{cnt}'_1, \mathsf{cnt}_1) \neq \bot\ \wedge\ b_1 = 0$: $/\!/\mathbf{G}_6$
83 $(epk', esk') \coloneqq \mathsf{KEM.KG}(; f_{\mathtt{ekg}}(\mathsf{cnt}_1))$ $/\!/\mathbf{G}_6$
84 **if** $epk = epk'$: $/\!/\mathbf{G}_6$
85 $d \coloneqq 1$, $k' \coloneqq \mathsf{KEM.Decaps}(esk', c)$ $/\!/\mathbf{G}_6$
86 **if** $k = k'$: $k^* \coloneqq f_{\mathtt{k}}(epk, c)$ $/\!/\mathbf{G}_6$
87 **if** $d = 0\ \wedge\ (b_2, u_2, v_2, \mathsf{cnt}'_2, \mathsf{cnt}_2) \neq \bot\ \wedge\ b_2 = 1$: $/\!/\mathbf{G}_6$
88 $(c', k') \coloneqq \mathsf{KEM.Encaps}(epk; f_{\mathtt{enc}}(epk, \mathsf{cnt}_2))$ $/\!/\mathbf{G}_6$
89 **if** $c = c'\ \wedge\ k = k'$: $k^* \coloneqq f_{\mathtt{k}}(epk, c)$ $/\!/\mathbf{G}_6$
90 **return** $f_{\mathtt{H}}(\mathsf{pk}_1, \mathsf{pk}_2, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k^*)$

Fig. 7. Games $\mathbf{G}_{[3\text{-}6]}$ for proving Theorem 1. For any party u, we have $(\mathsf{pk}_u, \mathsf{sk}_u) = \mathsf{GetKey}(u)$. Algorithm GetKey is the same as in Figure 6, and oracles $\textsc{Cor}$ and $\textsc{Pk}$ are the same as in Figure 4 and thus are omitted here.

Hence, if $\mathsf{CheckSid}(\mathsf{sID})$ returns 1, then sID is generated by $\textsc{Der}_{\mathrm{I}}$ or $\textsc{Der}_{\mathrm{R}}$ and has completed and has generated a session key $\neq \bot$ (except for the error probability above). This means that we can use $\mathsf{CheckSid}$ to check if a session identifier sID exists without recording the session identifier in the set $\mathcal{S}_{\mathtt{sid}}$

and check if sID has a valid session key without searching the list $\mathcal{L}_{\mathtt{key}}$. This observation allows this game to forget $\mathcal{S}_{\mathtt{sid}}$ and allows the next game to safely forget $\mathcal{L}_{\mathtt{key}}$.

From $\mathbf{G}_5$ onwards, we implicitly abort the game and return an independent uniform bit if $\mathsf{CheckSid}$ accepts a session identifier that has not been output by $\textsc{Der}_{\mathrm{I}}$ or $\textsc{Der}_{\mathrm{R}}$. The error probability is counted only once in the bound above. This convention is only for the game analysis, where we may keep the history needed to detect this event. The memory bounds of recording the history does not affect the statistical bounds.

Game $\mathbf{G}_6$ (Compute session keys without decapsulated keys). We modify H such that the session keys in $\mathsf{Der}_{\mathrm{I}}$ and $\mathsf{Der}_{\mathrm{R}}$ can be computed without using decapsulated KEM keys. To make the simulation consistent, we also change how session keys are computed in oracles $\mathsf{Der}_{\mathrm{I}}$ and $\mathsf{Der}_{\mathrm{R}}$, and use an auxiliary algorithm $\mathsf{ComputeSK}$ (cf. Figure 7) to compute the output session key of $\textsc{Test}$ and $\textsc{Rev}$.

Here we give a high-level description of the changes, and defer the details to Supp. Mat. F.4. In the H oracle in Figure 7, the code in lines 80 to 86 addresses the case when the ephemeral KEM public key epk is generated by some $\textsc{Init}_{\mathrm{I}}$ query (and hence we can recover back the corresponding esk and do decapsulation); the code in lines 87 to 89 addresses the case when the KEM ciphertext c is generated by some $\textsc{Der}_{\mathrm{R}}$ query (and hence we can recover back the corresponding k'). In either case, we can recover back the decapsulated symmetric key k and replace it with the output of $f_{\mathtt{k}}(epk, c)$ if the symmetric key k in the query matches the actual symmetric key k'. Then, we also change the oracles $\textsc{Test}$, $\textsc{Der}_{\mathrm{I}}$, and $\textsc{Der}_{\mathrm{R}}$ to consistently match the distribution of H via an internal auxiliary procedure $\mathsf{ComputeSK}(\mathsf{sID})$. Since we replace all "correct" decapsulated symmetric keys with outputs from $f_{\mathtt{k}}$, $\mathsf{ComputeSK}$ can compute the session key using the epk and c in the session identifier sID. All these changes are indistinguishable to $\mathcal{A}$ unless there is any decryption error of KEM or a collision of $f_{\mathtt{k}}$ happens in the query to H.

We summarize the analysis of $\mathbf{G}_6$ as a lemma and defer its proof to Supp. Mat. F.4.

Lemma 5. *With the assumptions as stated in Theorem 1, we have*

$$\left|\Pr\left[\mathbf{G}_5^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_6^{\mathcal{A}} \Rightarrow 1\right]\right| \leq (Q_{\mathtt{H}} + N_{\mathtt{ses}})^2 \cdot |\mathcal{K}|^{-1} + N_{\mathtt{ses}}^2 \cdot \delta_{\mathsf{KEM}}.$$

For games $\mathbf{G}_7$ and $\mathbf{G}_8$, we only present the changed oracles and new procedures in Figure 8, and the unchanged oracles and procedures are the same as in Figure 7. The goal of the game sequence $\mathbf{G}_7$-$\mathbf{G}_8$ is to identify whether $\mathcal{A}$ is tampering with messages from honest initiators or honest responders without maintaining the set $\mathcal{S}_{\mathtt{msg}}$.

Game $\mathbf{G}_7$ (Determine honest initiator messages). In oracle $\mathsf{Der}_{\mathrm{R}}$, we replace the if-condition $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$ with $\mathsf{CheckMsgI}(u, \mathsf{M}_u)$ (cf. line 28 in Figure 8), where $\mathsf{CheckMsgI}$ is an internal procedure defined in Figure 8 which checks if $\mathcal{A}$ faithfully delivers some protocol messages from $\textsc{Init}_{\mathrm{I}}$ to $\textsc{Der}_{\mathrm{R}}$. Moreover, after running $\mathcal{A}$, we also use $\mathsf{CheckMsgI}$ to replace $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$ in the procedure Valid (cf. lines 44 and 48).

We first summarize the effect of $\mathsf{CheckMsgI}$ in Lemma 6.

Lemma 6. *With the notations and assumptions as stated in Theorem 1, we have*

$$\mathsf{CheckMsgI}(u, (\alpha, epk, \sigma_1)) = 1 \iff (u, (\alpha, epk, \sigma_1)) \in \mathcal{S}_{\mathtt{msg}}$$

in $\mathbf{G}_7$*, except with probability at most* $2N_{\mathtt{ses}}^2|\mathcal{U}| \cdot 2^{-\ell}$.

Then we prove Lemma 6 as follows. On input $(u, \mathsf{M}_u = (\alpha, epk, \sigma_1))$, the procedure $\mathsf{CheckMsgI}$ tries recovering the initiator identity, ephemeral public key, and the signature of the session (cf. lines 00 to 05 in Figure 8) from the information hidden in α, and uses the random functions $f_{\mathtt{ekg}}$ and $f_{\mathtt{sI}}$ to check if the recovered information is consistent with the input (u, M_u). If so, then $\mathsf{CheckMsgI}$ returns 1, which indicates that $(u, (\alpha, epk, \sigma_1)) \in \mathcal{S}_{\mathtt{msg}}$.

It is straightforward to see that if $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$, then $\mathsf{CheckMsgI}(u, \mathsf{M}_u)$ always returns 1. So, using $\mathsf{CheckMsgI}$ never causes false negatives. To see how the adversary can cause false positives, we consider the following three cases. When $\mathcal{A}$ sends $(u, (\alpha, epk, \sigma_1)) \notin \mathcal{S}_{\mathtt{msg}}$ to $\textsc{Der}_{\mathrm{R}}$:

Case 1: α is forged by $\mathcal{A}$. In this case, α is not generated from either $\textsc{Init}_{\mathrm{I}}$ or $\textsc{Der}_{\mathrm{R}}$. Let $(b', u', v', \mathsf{cnt}'', \mathsf{cnt}') := h_{\mathtt{nce}}^{-1}(\alpha)$. To make $\mathsf{CheckMsgI}$ output 1, the inverse of the given α should fall into the set

$$\{(0, u, v, 0, \mathsf{cnt}') \mid u = u' \ \wedge \ v \in \mathcal{U} \ \wedge \ \mathsf{cnt}' \in [N_{\mathtt{ses}}]\}.$$

Alg $\mathsf{CheckMsgI}(u, \mathsf{M}_u = (\alpha, epk, \sigma_1))$

00 $(b', u', v, \mathsf{cnt}', \mathsf{cnt}) \coloneqq h_{\mathtt{nce}}^{-1}(\alpha)$
01 **if** $(b', u', v, \mathsf{cnt}', \mathsf{cnt}) \neq \bot \wedge u = u'$
$\quad \wedge\ \mathsf{cnt} \in [N_{\mathtt{ses}}] \wedge b' = 0 \wedge \mathsf{cnt}' = 0$
02 $\quad (epk', esk') \coloneqq \mathsf{KEM.KG}(; f_{\mathtt{ekg}}(\mathsf{cnt}))$
03 $\quad \sigma_1' \coloneqq \mathsf{Sign}(\mathsf{sk}_u, (u, \alpha, epk); f_{\mathtt{sI}}(u, \alpha, epk))$
04 $\quad$ **if** $epk' = epk \wedge \sigma_1' = \sigma_1$: **return** 1
05 **return** 0

Alg $\mathsf{CheckMsgR}(u, v, \mathsf{M}_u, \mathsf{M}_v)$

06 $(\alpha, epk, \sigma_1) \coloneqq \mathsf{M}_u$, $(\beta, c, \sigma_2) \coloneqq \mathsf{M}_v$
07 **if** $\neg\mathsf{CheckMsgI}(u, \mathsf{M}_u)$: **return** 0
08 $(b', u', v', \mathsf{cnt}_1', \mathsf{cnt}_2) \coloneqq h_{\mathtt{nce}}^{-1}(\beta)$
09 $(*, \mathsf{cnt}_1) \coloneqq h_{\mathtt{nce}}^{-1}(\alpha)$
10 **if** $(b', u', v', \mathsf{cnt}_1', \mathsf{cnt}_2) \neq \bot$
$\quad \wedge\ b' = 1 \wedge (u, v) = (u', v')$
$\quad \wedge\ \mathsf{cnt}_2 \in [N_{\mathtt{ses}}] \wedge \mathsf{cnt}_1' = \mathsf{cnt}_1$
11 $\quad y \coloneqq f_{\mathtt{enc}}(epk, \mathsf{cnt}_2)$
12 $\quad (c', k') \coloneqq \mathsf{KEM.Encaps}(epk; y)$
13 $\quad r_{\mathtt{sign}} \coloneqq f_{\mathtt{sR}}(u, v, \alpha, \beta, epk, c, \sigma_1)$
14 $\quad$ **if** $c' \neq c$: **return** 0
15 $\quad \sigma_2' \coloneqq \mathsf{Sign}(\mathsf{sk}_v, (u, v, \alpha, \beta, epk, c, \sigma_1); r_{\mathtt{sign}})$
16 $\quad$ **if** $\sigma_2' = \sigma_2$: **return** 1
17 **return** 0

Oracle $\textsc{Init}_{\mathrm{I}}(u, v)$

18 $\mathsf{cnt} \coloneqq \mathsf{cnt} + 1$, $\mathsf{seq}_1 \coloneqq h_{\mathtt{seq}}(u, v, \mathsf{cnt})$
19 $\alpha \coloneqq h_{\mathtt{nce}}(0, u, v, 0, \mathsf{cnt})$
20 $(epk, esk) \coloneqq \mathsf{KEM.KG}(; f_{\mathtt{ekg}}(\mathsf{cnt}))$
21 $\sigma \coloneqq \mathsf{Sign}(\mathsf{sk}_u, (u, \alpha, epk); f_{\mathtt{sI}}(u, \alpha, epk))$
22 $\mathsf{M}_u \coloneqq (\alpha, epk, \sigma)$
23 $\mathcal{S}_{\mathtt{msg}} \coloneqq \mathcal{S}_{\mathtt{msg}} \cup \{(u, \mathsf{M}_u)\}$ $\quad /\!\!/ \mathbf{G}_6$
24 **return** $(\mathsf{seq}_1, \mathsf{M}_u)$

Oracle $\textsc{Der}_{\mathrm{R}}(u, v, \mathsf{M}_u = (\alpha, epk, \sigma_1))$

25 **if** $\mathsf{Vrfy}(\mathsf{pk}_u, (u, \alpha, epk), \sigma_1) = 0$: **return** $\bot$
26 $\mathsf{cnt} \coloneqq \mathsf{cnt} + 1$
27 **if** $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$ $\quad /\!\!/ \mathbf{G}_6$
28 **if** $\mathsf{CheckMsgI}(u, \mathsf{M}_u)$ $\quad /\!\!/ \mathbf{G}_{[7\text{-}8]}$
29 $\quad (*, \mathsf{cnt}_1) \coloneqq h_{\mathtt{nce}}^{-1}(\alpha)$
30 $\quad \beta \coloneqq h_{\mathtt{nce}}(1, u, v, \mathsf{cnt}_1, \mathsf{cnt})$
31 **else** $\beta \coloneqq h_{\mathtt{nce}}(1, u, v, 0, \mathsf{cnt})$
32 $(c, k) \coloneqq \mathsf{KEM.Encaps}(epk; f_{\mathtt{enc}}(epk, \mathsf{cnt}))$
33 $r_{\mathtt{sign}} \coloneqq f_{\mathtt{sR}}(u, v, \alpha, \beta, epk, c, \sigma_1)$
34 $\sigma_2 \coloneqq \mathsf{Sign}(\mathsf{sk}_v, (u, v, \alpha, \beta, epk, c, \sigma_1); r_{\mathtt{sign}})$
35 $\mathsf{M}_v \coloneqq (\beta, c, \sigma_2)$
36 $\mathsf{sID} \coloneqq (\mathsf{seq} = f_{\mathtt{seq}}(u, v, \mathsf{M}_u, \mathsf{M}_v), u, v, \mathsf{M}_u, \mathsf{M}_v)$
37 **if** $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$ $\quad /\!\!/ \mathbf{G}_6$
38 **if** $\mathsf{CheckMsgI}(u, \mathsf{M}_u)$ $\quad /\!\!/ \mathbf{G}_{[7\text{-}8]}$
39 $\quad \mathcal{S}_{\mathtt{msg}} \coloneqq \mathcal{S}_{\mathtt{msg}} \cup \{(u, v, \mathsf{M}_u, \mathsf{M}_v)\}$ $\quad /\!\!/ \mathbf{G}_7$
40 $\mathcal{S}_{\mathtt{sid}} \coloneqq \mathcal{S}_{\mathtt{sid}} \cup \{\mathsf{sID}\}$
41 **return** sID

Alg $\mathsf{Valid}(\mathsf{sID})$

42 **if** $\mathsf{sID} = (\mathsf{seq}, u, v, \mathsf{M}_u, \mathsf{M}_v)$
43 $\quad$ **if** $u \notin \mathcal{S}_{\mathtt{cor}} \vee (u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$ $\quad /\!\!/ \mathbf{G}_6$
44 $\quad$ **if** $u \notin \mathcal{S}_{\mathtt{cor}} \vee \mathsf{CheckMsgI}(u, \mathsf{M}_u)$ $\quad /\!\!/ \mathbf{G}_{[7\text{-}8]}$
45 $\quad\quad$ **return** 1
46 **if** $\mathsf{sID} = (\mathsf{seq}_1, \mathsf{seq}_2, u, v, \mathsf{M}_u, \mathsf{M}_v)$
47 $\quad$ **if** $v \notin \mathcal{S}_{\mathtt{cor}} \vee (u, v, \mathsf{M}_u, \mathsf{M}_v) \in \mathcal{S}_{\mathtt{msg}}$ $\quad /\!\!/ \mathbf{G}_{[6\text{-}7]}$
48 $\quad$ **if** $v \notin \mathcal{S}_{\mathtt{cor}} \vee \mathsf{CheckMsgR}(u, v, \mathsf{M}_u, \mathsf{M}_v)$ $\quad /\!\!/ \mathbf{G}_8$
49 $\quad\quad$ **return** 1
50 **return** 0

Fig. 8. Oracles $\textsc{Der}_{\mathrm{I}}$ and $\textsc{Der}_{\mathrm{R}}$, internal procedures $\mathsf{CheckMsgI}$, $\mathsf{CheckMsgR}$, and Valid of $\mathbf{G}_{[6\text{-}8]}$. Other oracles and procedures are the same as in Figure 7. Since $\mathbf{G}_6$, all session keys are computed by $\mathsf{ComputeSK}$ and the list $\mathcal{L}_{\mathtt{SK}}$ is no longer needed, so we do not explicitly calculate session keys in the code.

There are $|\mathcal{U}| \cdot N_{\mathtt{ses}}$ tuples in the set. Since $h_{\mathtt{nce}}$ is a random injective function, the co-domain of this set (under $h_{\mathtt{nce}}$) is a random subset of the range of $h_{\mathtt{nce}}$ with size 2^ℓ. Hence, for arbitrary α, the probability that $h_{\mathtt{nce}}^{-1}(\alpha)$ falls into the above set is $\frac{|\mathcal{U}| \cdot N_{\mathtt{ses}}}{2^\ell}$. Since $\mathcal{A}$ has seen at most $N_{\mathtt{ses}}$ points of the image of $h_{\mathtt{nce}}$, the probability that $\mathcal{A}$ successfully forges such a α is bounded by $\frac{|\mathcal{U}| \cdot N_{\mathtt{ses}}}{2^\ell - N_{\mathtt{ses}}} \leq \frac{2|\mathcal{U}| N_{\mathtt{ses}}}{2^\ell}$ (since by the assumption of Theorem 1, $2^\ell/2 > N_{\mathtt{ses}}$). Then, as $\mathcal{A}$ can issue at most $N_{\mathtt{ses}}$ queries to $\textsc{Der}_{\mathrm{R}}$, the probability that $\mathcal{A}$ made at least one successful forgery of such α is bounded by $2N_{\mathtt{ses}}^2 |\mathcal{U}| \cdot 2^{-\ell}$.

Case 2: α is from $\textsc{Init}_{\mathrm{I}}(u, *)$. In this case, $\mathcal{A}$ replays α from $\textsc{Init}_{\mathrm{I}}(u, v)$ for some v, but with a different epk or σ_1. Let $(\alpha, epk', \sigma_1') \neq (\alpha, epk, \sigma_1)$ be the original initiator protocol message that corresponds to α (namely, $(u, (\alpha, epk', \sigma_1')) \in \mathcal{S}_{\mathtt{msg}}$). Let $(0, u, v, 0, \mathsf{cnt}') \coloneqq h_{\mathtt{nce}}^{-1}(\alpha)$, $x \coloneqq f_{\mathtt{ekg}}(\mathsf{cnt}')$, and $(epk', esk') \coloneqq \mathsf{KEM.KG}(; x)$. If $epk \neq epk'$, then $\mathsf{CheckMsgI}$ returns 0. So, to make $\mathsf{CheckMsgI}$ output 1, we must have $epk = epk'$, which means $\sigma_1 \neq \sigma_1'$. However, since $\sigma_1' = \mathsf{Sign}(\mathsf{sk}_u, (u, \alpha, epk); f_{\mathtt{sI}}(u, \alpha, epk))$ (by line 31 in Figure 8), if $\sigma_1 \neq \sigma_1'$, then $\mathsf{CheckMsgI}(u, \alpha, epk, \sigma_1)$ still returns 0. Hence, this case never causes false positives.

Finally, replaying α from $\textsc{Der}_{\mathrm{R}}$ (via letting $\alpha \coloneqq \beta$) never helps because the label input of $h_{\mathtt{nce}}^{-1}(\beta)$ is always 1, while the label input of $h_{\mathtt{nce}}^{-1}(\alpha)$ is always 0 if α is from $\textsc{Init}_{\mathrm{I}}$.

Therefore, $\mathsf{CheckMsgI}(u, \alpha, epk, \sigma_1)$ returns 1 if and only if $(u, (\alpha, epk, \sigma_1)) \in \mathcal{S}_{\mathtt{msg}}$ except with probability at most $2N_{\mathtt{ses}}^2 |\mathcal{U}| \cdot 2^{-\ell}$. Therefore, we have

$$\left|\Pr\left[\mathbf{G}_6^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_7^{\mathcal{A}} \Rightarrow 1\right]\right| \leq \frac{2N_{\mathtt{ses}}^2 |\mathcal{U}|}{2^\ell}$$

Game $\mathbf{G}_8$ (Determine passive sessions). In this game, we replace the if-condition $(u, v, \mathsf{M}_u, \mathsf{M}_v) \in \mathcal{S}_{\mathtt{msg}}$ in the procedure Valid with a new internal procedure CheckMsgR. In $\mathbf{G}_7$, $(u, v, \mathsf{M}_u, \mathsf{M}_v) \in \mathcal{S}_{\mathtt{msg}}$ means that there exists a responder session of v that received the initiator message M_u from u and sent the responder message M_v, and CheckMsgR is defined to check this condition without maintaining $\mathcal{S}_{\mathtt{msg}}$. We first summarize the effect of introducing CheckMsgR.

Lemma 7. *With the notations and assumptions as stated in $\mathbf{G}_8$ of the proof of Theorem 1, we have $(u, v, \mathsf{M}_u, \mathsf{M}_v) \in \mathcal{S}_{\mathtt{msg}} \iff \mathsf{CheckMsgR}(u, v, \mathsf{M}_u, \mathsf{M}_v) = 1$ except with probability at most $2N_{\mathtt{ses}}^2(|\mathcal{U}| + 1) \cdot 2^{-\ell}$.*

The analysis of Lemma 7 is more subtle than that of Lemma 6, as CheckMsgR needs to check if M_v is a response to M_u (not just a message from some honest responder session). The detailed proof is provided in Supp. Mat. F.5. Here we give a proof sketch of Lemma 7.

On input $(u, v, \mathsf{M}_u = (\alpha, epk, \sigma_1), \mathsf{M}_v = (\beta, c, \sigma_2))$, the procedure CheckMsgR first checks whether $\mathsf{CheckMsgI}(u, \mathsf{M}_u) = 1$ to ensure that $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$ (i.e., M_u is an honest initiator message from some user u). Then, it checks if (u, v, M_v) is consistent with β using a similar method as CheckMsgI (i.e., recovering the ephemeral encapsulation ciphertext and signature from β and checking if they are consistent with M_v).

CheckMsgR also checks if the β value matches α (i.e., β's preimage includes α's counter value) to ensure that M_v is a response to M_u. This step is for preventing the adversary from using some valid (u, M_u) and valid (v, M_v) to make CheckMsgR return 1 without v actually responding to u's message M_u.

By Lemma 7, we have

$$\left|\Pr\left[\mathbf{G}_7^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_8^{\mathcal{A}} \Rightarrow 1\right]\right| \leq 2N_{\mathtt{ses}}^2(|\mathcal{U}| + 1) \cdot 2^{-\ell}$$

Game $\mathbf{G}_9$ (Exclude wFS attacks). We introduce an abort event in oracle H. When $\mathcal{A}$ queries H on $(\mathsf{pk}_1, \mathsf{pk}_2, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k)$, if the query corresponds to a matching initiator and responder session (i.e., CheckMsgR returns 1, cf. lines 04 to 07 in Figure 9) and the encapsulated key k is correct (cf. lines 08 to 13 in Figure 9), then we raise the flag $\mathtt{Flag}_{\mathtt{queryH}}$, abort the game, and return a random bit. This game is for excluding the wFS attack, i.e., $\mathcal{A}$ cannot distinguish session keys of passive honest sessions from random keys.

We summarize the effect of introducing the above abort event as follows. The detailed proof is given in Supp. Mat. F.6.

Lemma 8. *With the notations and assumptions as stated in $\mathbf{G}_9$ of the proof of Theorem 1, there exists an adversary $\mathcal{B}_{\mathsf{KEM}}$ against the OW-PCA security of KEM in the multi-user, multi-challenge setting such that*

$$\left|\Pr\left[\mathbf{G}_8^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_9^{\mathcal{A}} \Rightarrow 1\right]\right| \leq \Pr[\mathsf{OW\text{-}PCA}_{\mathsf{KEM}}^{\mathcal{B}_{\mathsf{KEM}}} \Rightarrow 1] + 2Q_{\mathtt{H}}N_{\mathtt{ses}}(|\mathcal{U}| + 1) \cdot 2^{-\ell},$$

and the running time and reduced memory usage of $\mathcal{B}_{\mathsf{KEM}}$ are about the same as those of $\mathcal{A}$. More specifically, we have

$$\begin{aligned}
\mathbf{T}^*(\mathcal{B}_{\mathsf{KEM}}) &= \mathbf{T}(\mathcal{A}) + \mathrm{O}(Q_{\mathtt{query}} + Q_{\mathtt{H}}) \cdot \mathbf{T}(\mathtt{aSigKEM}) \\
\mathbf{Mem}^*(\mathcal{B}_{\mathsf{KEM}}) &= \mathbf{Mem}(\mathcal{A}) + \mathrm{O}(\log N_{\mathtt{ses}}) + \mathrm{O}(\ell) \\
&\quad + \mathrm{O}(\log |\mathcal{U}|) + \mathrm{O}(\log |\mathcal{SEQ}|) + \mathbf{Mem}(\mathtt{aSigKEM})
\end{aligned}$$

Here we give a high-level intuition of the proof of Lemma 8. The abort event happens (i.e., $\mathtt{Flag}_{\mathtt{queryH}}$ is raised) when $\mathcal{A}$ finds the decapsulated symmetric key k of (epk, c) when it queries H on the tuple corresponding to the matching initiator session of epk and the responder session of c. Since this event is only relevant to honestly generated (epk, c) (i.e., those generated by KEM.KG in $\textsc{Init}_{\mathrm{I}}$ and KEM.Encaps in $\textsc{Der}_{\mathrm{R}}$), we can use this event to solve the OW-PCA problem of KEM by simulating the game for $\mathcal{A}$. This is done by a reduction $\mathcal{B}_{\mathsf{KEM}}$ (cf. Figure 19) , which embeds the OW-PCA challenge public keys and ciphertexts into the $\textsc{Init}_{\mathrm{I}}$ and $\textsc{Der}_{\mathrm{R}}$ oracles (and uses counters to index these challenge public keys and ciphertexts). It uses the $\textsc{Pco}$ oracle to replace all $k' = k$ predicates appearing in CheckMsgI, CheckMsgR, and H. Note that in this game, we allow $\mathcal{A}$ to (indirectly) query CheckMsgR via issuing H queries. By applying Lemma 7 with the union bound over all H queries, the error probability (i.e., producing false positives from CheckMsgR) is at most $2Q_{\mathtt{H}}N_{\mathtt{ses}}(|\mathcal{U}| + 1) \cdot 2^{-\ell}$.

Oracle $\mathsf{H}(\mathsf{pk}_1, \mathsf{pk}_2, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k)$

00 $k^* \coloneqq k$, $d \coloneqq 0$
01 $(b_1, u_1, v_1, \mathsf{cnt}_1', \mathsf{cnt}_1) \coloneqq h_{\mathtt{nce}}^{-1}(\alpha)$
02 $(b_2, u_2, v_2, \mathsf{cnt}_2', \mathsf{cnt}_2) \coloneqq h_{\mathtt{nce}}^{-1}(\beta)$
03 $\mathsf{M}_u \coloneqq (\alpha, epk, \sigma_1)$, $\mathsf{M}_v \coloneqq (\beta, c, \sigma_2)$
04 **if** $h_{\mathtt{nce}}^{-1}(\alpha) \neq \bot \wedge h_{\mathtt{nce}}^{-1}(\beta) \neq \bot$ // $\mathbf{G}_9$-$\mathbf{G}_{10}$
05 $\quad \wedge\ \mathsf{pk}_1 = \textsc{Pk}(u_1)$ // $\mathbf{G}_9$-$\mathbf{G}_{10}$
06 $\quad \wedge\ \mathsf{pk}_2 = \textsc{Pk}(v_2)$ // $\mathbf{G}_9$-$\mathbf{G}_{10}$
07 $\quad \wedge\ \mathsf{CheckMsgR}(u_1, v_2, \mathsf{M}_u, \mathsf{M}_v)$ // $\mathbf{G}_9$-$\mathbf{G}_{10}$
08 $\quad x \coloneqq f_{\mathtt{ekg}}(\mathsf{cnt}_1)$ // $\mathbf{G}_9$-$\mathbf{G}_{10}$
09 $\quad y \coloneqq f_{\mathtt{enc}}(epk, \mathsf{cnt}_2)$ // $\mathbf{G}_9$-$\mathbf{G}_{10}$
10 $\quad (epk', esk') \coloneqq \mathsf{KEM.KG}(; x)$ // $\mathbf{G}_9$-$\mathbf{G}_{10}$
11 $\quad (c', k') \coloneqq \mathsf{KEM.Encaps}(epk; y)$ // $\mathbf{G}_9$-$\mathbf{G}_{10}$
12 $\quad k'' \coloneqq \mathsf{KEM.Decaps}(esk', c)$ // $\mathbf{G}_9$-$\mathbf{G}_{10}$
13 $\quad$ **if** $(epk, c, k) = (epk', c', k'')$ // $\mathbf{G}_9$-$\mathbf{G}_{10}$
14 $\quad\quad \mathtt{Flag}_{\mathtt{queryH}} \coloneqq$ **true** // $\mathbf{G}_9$-$\mathbf{G}_{10}$
15 $\quad\quad$ **abort** // $\mathbf{G}_9$-$\mathbf{G}_{10}$
16 $\quad\quad$ **output** $b \leftarrow \{0,1\}$ // $\mathbf{G}_9$-$\mathbf{G}_{10}$
17 **if** $h_{\mathtt{nce}}^{-1}(\alpha) \neq \bot\ \wedge\ b_1 = 0$:
18 $\quad (epk', esk') \coloneqq \mathsf{KEM.KG}(; f_{\mathtt{ekg}}(\mathsf{cnt}_1))$
19 $\quad$ **if** $epk = epk'$:
20 $\quad\quad d \coloneqq 1$, $k' \coloneqq \mathsf{KEM.Decaps}(esk', c)$
21 $\quad\quad$ **if** $k = k'$: $k^* \coloneqq f_{\mathtt{k}}(epk, c)$
22 **if** $d = 0\ \wedge\ h_{\mathtt{nce}}^{-1}(\beta) \neq \bot\ \wedge\ b_2 = 1$:
23 $\quad y \coloneqq f_{\mathtt{enc}}(epk, \mathsf{cnt}_2)$
24 $\quad (c', k') \coloneqq \mathsf{KEM.Encaps}(epk; y)$
25 $\quad$ **if** $c = c'\ \wedge\ k = k'$: $k^* \coloneqq f_{\mathtt{k}}(epk, c)$
26 **return** $f_{\mathtt{H}}(\mathsf{pk}_1, \mathsf{pk}_2, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k^*)$

Oracle $\textsc{Der}_{\mathrm{R}}(u, v, \mathsf{M}_u = (\alpha, epk, \sigma_1))$

27 **if** $\mathsf{Vrfy}(\mathsf{pk}_u, (u, \alpha, epk), \sigma_1) = 0$: **return** $\bot$
28 **if** $\neg\mathsf{CheckMsgI}(u, \mathsf{M}_u)$ // $\mathbf{G}_{10}$
29 $\quad$ **if** $u \notin \mathcal{S}_{\mathtt{cor}}$ // $\mathbf{G}_{10}$
30 $\quad\quad \mathtt{Flag}_{\mathtt{forge}} \coloneqq$ **true** // $\mathbf{G}_{10}$
31 $\quad\quad$ **abort** and **output** $b \leftarrow \{0,1\}$ // $\mathbf{G}_{10}$
32 $\mathsf{cnt} \coloneqq \mathsf{cnt} + 1$
33 **if** $\mathsf{CheckMsgI}(u, \mathsf{M}_u)$
34 $\quad (*, \mathsf{cnt}_1) \coloneqq h_{\mathtt{nce}}^{-1}(\alpha)$
35 $\quad \beta \coloneqq h_{\mathtt{nce}}(1, u, v, \mathsf{cnt}_1, \mathsf{cnt})$
36 **else** $\beta \coloneqq h_{\mathtt{nce}}(1, u, v, 0, \mathsf{cnt})$
37 $(c, k) \coloneqq \mathsf{KEM.Encaps}(epk; f_{\mathtt{enc}}(epk, \mathsf{cnt}))$
38 $r_{\mathtt{sign}} \coloneqq f_{\mathtt{sR}}(u, v, \alpha, \beta, epk, c, \sigma_1)$
39 $\sigma_2 \coloneqq \mathsf{Sign}(\mathsf{sk}_v, (u, v, \alpha, \beta, epk, c, \sigma_1); r_{\mathtt{sign}})$
40 $\mathsf{M}_v \coloneqq (\beta, c, \sigma_2)$
41 $\mathsf{sID} \coloneqq (\mathsf{seq} = f_{\mathtt{seq}}(u, v, \mathsf{M}_u, \mathsf{M}_v), u, v, \mathsf{M}_u, \mathsf{M}_v)$
42 $\mathcal{S}_{\mathtt{sid}} \coloneqq \mathcal{S}_{\mathtt{sid}} \cup \{\mathsf{sID}\}$
43 **return** sID

Oracle $\textsc{Der}_{\mathrm{I}}(u, v, \mathsf{seq}_1, \mathsf{M}_v = (\beta, c, \sigma_2))$

44 $(u', v', \mathsf{cnt}') \coloneqq h_{\mathtt{seq}}^{-1}(\mathsf{seq}_1)$
45 **if** $(u', v') \neq (u, v) \vee \mathsf{cnt} < \mathsf{cnt}'$: **return** $\bot$
46 $\alpha \coloneqq h_{\mathtt{nce}}(0, u, v, 0, \mathsf{cnt}')$
47 $(epk, esk) \coloneqq \mathsf{KEM.KG}(; f_{\mathtt{ekg}}(\mathsf{cnt}'))$
48 $\sigma_1 \coloneqq \mathsf{Sign}(\mathsf{sk}_u, (u, \alpha, epk); f_{\mathtt{sI}}(u, \alpha, epk))$
49 $\mathsf{M}_u \coloneqq (\alpha, epk, \sigma_1)$
50 **if** $\mathsf{Vrfy}(\mathsf{pk}_v, (u, v, \alpha, \beta, epk, c, \sigma_1), \sigma_2) = 0$:
51 $\quad$ **return** $\bot$
52 **if** $\neg\mathsf{CheckMsgR}(u, v, \mathsf{M}_u, \mathsf{M}_v)$ // $\mathbf{G}_{10}$
53 $\quad$ **if** $v \notin \mathcal{S}_{\mathtt{cor}}$ // $\mathbf{G}_{10}$
54 $\quad\quad \mathtt{Flag}_{\mathtt{forge}} \coloneqq$ **true** // $\mathbf{G}_{10}$
55 $\quad\quad$ **abort** and **output** $b \leftarrow \{0,1\}$ // $\mathbf{G}_{10}$
56 $\mathsf{seq}_2 \coloneqq f_{\mathtt{seq}}(\mathsf{seq}_1, u, v, \mathsf{M}_u, \mathsf{M}_v)$
57 $\mathsf{sID} \coloneqq (\mathsf{seq}_1, \mathsf{seq}_2, u, v, \mathsf{M}_u, \mathsf{M}_v)$
58 $\mathcal{S}_{\mathtt{sid}} \coloneqq \mathcal{S}_{\mathtt{sid}} \cup \{\mathsf{sID}\}$
59 **return** sID

Fig. 9. Oracles $\textsc{Der}_{\mathrm{I}}$, $\textsc{Der}_{\mathrm{R}}$, and H of $\mathbf{G}_{[8\text{-}10]}$. For readability, we only present changed oracles. Other oracles are the same as in Figure 7. Since $\mathbf{G}_6$, all session keys are computed by $\mathsf{ComputeSK}$ and the list $\mathcal{L}_{\mathtt{SK}}$ is no longer needed, so we do not explicitly calculate session keys in $\textsc{Der}_{\mathrm{I}}$ and $\textsc{Der}_{\mathrm{R}}$ in these games.

Game $\mathbf{G}_{10}$ (Exclude no-corruption active attacks). Finally, if $\mathcal{A}$ forges any valid fresh signature of any uncorrupted user, we then raise $\mathtt{Flag}_{\mathtt{forge}}$ to **true**, abort the game, and return a random bit (cf. lines 52 to 55 and lines 29 to 31 in Figure 9). This game ensures that $\mathcal{A}$ cannot impersonate any uncorrupted user.

Let **Forge** be the event that $\mathtt{Flag}_{\mathtt{forge}}$ is raised as **true** at any point of the game. We construct a forger $\mathcal{B}_{\mathsf{SIG}}$ against the MU^c-sEUF-1CMA security of SIG to bound $\Pr[\mathbf{Forge}]$. Intuitively, the occurrence of **Forge** means that there is a valid signature that is not from any honest session of the uncorrupted user, which corresponds to a forgery of the signature scheme. We summarize the probability difference introduced by this event as follows; a detailed proof can be found in Supp. Mat. F.7.

Lemma 9. *With the notations and assumptions as stated in $\mathbf{G}_9$ of the proof of Theorem 1, there exists an adversary $\mathcal{B}_{\mathsf{SIG}}$ against the MU^c-sEUF-1CMA security of SIG such that*

$$\left|\Pr\left[\mathbf{G}_9^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_{10}^{\mathcal{A}} \Rightarrow 1\right]\right| \leq \Pr[\mathsf{MU}^c\text{-}\mathsf{sEUF\text{-}1CMA}_{\mathsf{SIG}}^{\mathcal{B}_{\mathsf{SIG}}} \Rightarrow 1]$$

and the running time and reduced memory usage of $\mathcal{B}_{\mathsf{SIG}}$ are about the same as those of $\mathcal{A}$. More specifically, we have

$$\begin{aligned}\mathbf{Mem}^*(\mathcal{B}_{\mathsf{SIG}}) &= \mathbf{Mem}(\mathcal{A}) + \mathrm{O}(\log N_{\mathtt{ses}}) + \mathrm{O}(\ell) \\ &\quad + \mathrm{O}(\log|\mathcal{U}|) + \mathrm{O}(\log|\mathcal{SEQ}|) + \mathbf{Mem}(\mathtt{aSigKEM}) \\ \mathbf{T}^*(\mathcal{B}_{\mathsf{SIG}}) &= \mathbf{T}(\mathcal{A}) + \mathrm{O}(Q_{\mathtt{query}} + Q_{\mathtt{H}}) \cdot \mathbf{T}(\mathtt{aSigKEM})\end{aligned}$$

Analysis of $\mathbf{G}_{10}$. For fixed users and messages, the injective nonce generation allows at most one responder session and fixes the initial state of an initiator session. Repeated calls to DER_I with this state and the same responder message return the same session identifier. Hence, each session has at most one matching session. Consider any valid and fresh test session sID^*. Freshness excludes Rev queries on sID^* and its matching session. Hence, in the random oracle model, $\mathcal{A}$ must query H on the corresponding tuple to distinguish whether the key from $\text{Test}(\mathsf{sID}^*)$ is real or random.

Let $(u, v, \mathsf{M}_u, \mathsf{M}_v)$ be the users-messages tuple corresponding to sID^*. We distinguish two cases:

- sID^* is an initiator session. By Valid (cf. Figure 8), either $\mathsf{CheckMsgR}(u, v, \mathsf{M}_u, \mathsf{M}_v) = 1$ or $v \notin \mathcal{S}_{\mathtt{cor}}$. In the first case, a subsequent H query on the corresponding tuple raises **QueryH**, so the game aborts and returns a random bit. Otherwise, v is uncorrupted and completing sID^* raises **Forge** in DER_I, so the game aborts before the session is completed.
- sID^* is a responder session. Similarly, Valid requires $\mathsf{CheckMsgI}(u, \mathsf{M}_u) = 1$ or $u \notin \mathcal{S}_{\mathtt{cor}}$. In the first case, DER_R generates M_v such that $\mathsf{CheckMsgR}(u, v, \mathsf{M}_u, \mathsf{M}_v) = 1$ (cf. Figure 9), and the same **QueryH** argument applies. Otherwise, completing the session raises **Forge** in DER_R.

For distinct session-key inputs, $f_{\mathtt{H}}$ and $f_{\mathtt{rSK}}$ have independent uniform outputs, so keys of other sessions do not help. It remains to count in two lucky events: $\mathcal{A}$ queries H on the corresponding tuple before sID^* is generated, or guesses $f_{\mathtt{k}}(epk, c)$ as the key in a query to H. Each session has a random nonce, and $f_{\mathtt{k}}$ is a random function. Taking a union bound over at most $Q_{\mathtt{H}}$ queries and $N_{\mathtt{ses}}$ sessions bounds these probabilities by $Q_{\mathtt{H}} N_{\mathtt{ses}} \cdot 2^{-\ell}$ and $Q_{\mathtt{H}} N_{\mathtt{ses}} \cdot |\mathcal{K}|^{-1}$, respectively. Except with these probabilities, real and random test keys have the same distribution from $\mathcal{A}$'s view. Hence, we have

$$\Pr\left[\mathbf{G}_{10}^{\mathcal{A}} \Rightarrow 1\right] = \frac{1}{2} + \frac{Q_{\mathtt{H}} N_{\mathtt{ses}}}{2^{\ell}} + \frac{Q_{\mathtt{H}} N_{\mathtt{ses}}}{|\mathcal{K}|}$$

This concludes the proof of Theorem 1 by combining the above probability differences.

Acknowledgments. We thank the anonymous reviewers of Asiacrypt 2026, Crypto 2026, and Eurocrypt 2026 for their valuable comments and suggestions, which pointed out several proof gaps in our previous version and provided helpful feedback on the technical aspects and presentation of our work. We have fixed these gaps in this version.

We used ChatGPT and OpenAI Codex for manuscript editing and consistency checks, reviewed their suggested and implemented changes, and take responsibility for the final content. Further details are provided in Supp. Mat. G.

Jiaxin Pan was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 572525990.

References

ABR01. Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In David Naccache, editor, CT-RSA 2001, volume 2020 of LNCS, pages 143–158. Springer, Berlin, Heidelberg, April 2001.

ACFK17. Benedikt Auerbach, David Cash, Manuel Fersch, and Eike Kiltz. Memory-tight reductions. In Jonathan Katz and Hovav Shacham, editors, CRYPTO 2017, Part I, volume 10401 of LNCS, pages 101–132. Springer, Cham, August 2017.

APS15. Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. Journal of Mathematical Cryptology, 9(3):169–203, 2015.

BDGL16. Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In Robert Krauthgamer, editor, 27th SODA, pages 10–24. ACM-SIAM, January 2016.

Bha20. Rishiraj Bhattacharyya. Memory-tight reductions for practical key encapsulation mechanisms. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, PKC 2020, Part I, volume 12110 of LNCS, pages 249–278. Springer, Cham, May 2020.

BHJ+15. Christoph Bader, Dennis Hofheinz, Tibor Jager, Eike Kiltz, and Yong Li. Tightly-secure authenticated key exchange. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, TCC 2015, Part I, volume 9014 of LNCS, pages 629–658. Springer, Berlin, Heidelberg, March 2015.

BKW03. Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. J. ACM, 50(4):506–519, July 2003.

BR96. Mihir Bellare and Phillip Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. In Ueli M. Maurer, editor, EUROCRYPT'96, volume 1070 of LNCS, pages 399–416. Springer, Berlin, Heidelberg, May 1996.

BR06. Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, EUROCRYPT 2006, volume 4004 of LNCS, pages 409–426. Springer, Berlin, Heidelberg, May / June 2006.

BR09. Mihir Bellare and Thomas Ristenpart. Simulation without the artificial abort: Simplified proof and improved concrete security for Waters' IBE scheme. In Antoine Joux, editor, EUROCRYPT 2009, volume 5479 of LNCS, pages 407–424. Springer, Berlin, Heidelberg, April 2009.

CN11. Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, ASIACRYPT 2011, volume 7073 of LNCS, pages 1–20. Springer, Berlin, Heidelberg, December 2011.

DGJL21a. Denis Diemert, Kai Gellert, Tibor Jager, and Lin Lyu. Digital signatures with memory-tight security in the multi-challenge setting. In Mehdi Tibouchi and Huaxiong Wang, editors, ASIACRYPT 2021, Part IV, volume 13093 of LNCS, pages 403–433. Springer, Cham, December 2021.

DGJL21b. Denis Diemert, Kai Gellert, Tibor Jager, and Lin Lyu. More efficient digital signatures with tight multi-user security. In Juan Garay, editor, PKC 2021, Part II, volume 12711 of LNCS, pages 1–31. Springer, Cham, May 2021.

EKM17. Andre Esser, Robert Kübler, and Alexander May. LPN decoded. In Jonathan Katz and Hovav Shacham, editors, CRYPTO 2017, Part II, volume 10402 of LNCS, pages 486–514. Springer, Cham, August 2017.

FKP17. Manuel Fersch, Eike Kiltz, and Bertram Poettering. On the one-per-message unforgeability of (EC)DSA and its variants. In Yael Kalai and Leonid Reyzin, editors, TCC 2017, Part II, volume 10678 of LNCS, pages 519–534. Springer, Cham, November 2017.

FSXY12. Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. Strongly secure authenticated key exchange from factoring, codes, and lattices. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, PKC 2012, volume 7293 of LNCS, pages 467–484. Springer, Berlin, Heidelberg, May 2012.

FSXY13. Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. Practical and post-quantum authenticated key exchange from one-way secure key encapsulation mechanism. In Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng, editors, ASIACCS 13, pages 83–94. ACM Press, May 2013.

GGJJ23. Kai Gellert, Kristian Gjøsteen, Håkon Jacobsen, and Tibor Jager. On optimal tightness for key exchange with full forward secrecy via key confirmation. In Helena Handschuh and Anna Lysyanskaya, editors, CRYPTO 2023, Part IV, volume 14084 of LNCS, pages 297–329. Springer, Cham, August 2023.

GGJT22. Ashrujit Ghoshal, Riddhi Ghosal, Joseph Jaeger, and Stefano Tessaro. Hiding in plain sight: Memory-tight proofs via randomness programming. In Orr Dunkelman and Stefan Dziembowski, editors, EUROCRYPT 2022, Part II, volume 13276 of LNCS, pages 706–735. Springer, Cham, May / June 2022.

GHKW16. Romain Gay, Dennis Hofheinz, Eike Kiltz, and Hoeteck Wee. Tightly CCA-secure encryption without pairings. In Marc Fischlin and Jean-Sébastien Coron, editors, EUROCRYPT 2016, Part I, volume 9665 of LNCS, pages 1–27. Springer, Berlin, Heidelberg, May 2016.

GJ18. Kristian Gjøsteen and Tibor Jager. Practical and tightly-secure digital signatures and authenticated key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, CRYPTO 2018, Part II, volume 10992 of LNCS, pages 95–125. Springer, Cham, August 2018.

GJT20. Ashrujit Ghoshal, Joseph Jaeger, and Stefano Tessaro. The memory-tightness of authenticated encryption. In Daniele Micciancio and Thomas Ristenpart, editors, CRYPTO 2020, Part I, volume 12170 of LNCS, pages 127–156. Springer, Cham, August 2020.

GT20. Ashrujit Ghoshal and Stefano Tessaro. On the memory-tightness of hashed ElGamal. In Anne Canteaut and Yuval Ishai, editors, EUROCRYPT 2020, Part II, volume 12106 of LNCS, pages 33–62. Springer, Cham, May 2020.

HHK17. Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, TCC 2017, Part I, volume 10677 of LNCS, pages 341–371. Springer, Cham, November 2017.

HHM22. Kathrin Hövelmanns, Andreas Hülsing, and Christian Majenz. Failing gracefully: Decryption failures and the Fujisaki-Okamoto transform. In Shweta Agrawal and Dongdai Lin, editors, ASIACRYPT 2022, Part IV, volume 13794 of LNCS, pages 414–443. Springer, Cham, December 2022.

HJ12. Dennis Hofheinz and Tibor Jager. Tightly secure signatures and public-key encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, CRYPTO 2012, volume 7417 of LNCS, pages 590–607. Springer, Berlin, Heidelberg, August 2012.

HK17. Gottfried Herold and Elena Kirshanova. Improved algorithms for the approximate k-list problem in euclidean norm. In Serge Fehr, editor, PKC 2017, Part I, volume 10174 of LNCS, pages 16–40. Springer, Berlin, Heidelberg, March 2017.

HKM18. Gottfried Herold, Elena Kirshanova, and Alexander May. On the asymptotic complexity of solving LWE. DCC, 86(1):55–83, 2018.

HKSU20. Kathrin Hövelmanns, Eike Kiltz, Sven Schäge, and Dominique Unruh. Generic authenticated key exchange in the quantum random oracle model. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, PKC 2020, Part II, volume 12111 of LNCS, pages 389–422. Springer, Cham, May 2020.

HLG23. Shuai Han, Shengli Liu, and Dawu Gu. Almost tight multi-user security under adaptive corruptions & leakages in the standard model. In Carmit Hazay and Martijn Stam, editors, EUROCRYPT 2023, Part III, volume 14006 of LNCS, pages 132–162. Springer, Cham, April 2023.

JK22. Joseph Jaeger and Akshaya Kumar. Memory-tight multi-challenge security of public-key encryption. In Shweta Agrawal and Dongdai Lin, editors, ASIACRYPT 2022, Part III, volume 13793 of LNCS, pages 454–484. Springer, Cham, December 2022.

JKRS21. Tibor Jager, Eike Kiltz, Doreen Riepel, and Sven Schäge. Tightly-secure authenticated key exchange, revisited. In Anne Canteaut and François-Xavier Standaert, editors, EUROCRYPT 2021, Part I, volume 12696 of LNCS, pages 117–146. Springer, Cham, October 2021.

JZC+18. Haodong Jiang, Zhenfeng Zhang, Long Chen, Hong Wang, and Zhi Ma. IND-CCA-secure key encapsulation mechanism in the quantum random oracle model, revisited. In Hovav Shacham and Alexandra Boldyreva, editors, CRYPTO 2018, Part III, volume 10993 of LNCS, pages 96–125. Springer, Cham, August 2018.

LDK+20. Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

LLGW20. Xiangyu Liu, Shengli Liu, Dawu Gu, and Jian Weng. Two-pass authenticated key exchange with explicit authentication and tight security. In Shiho Moriai and Huaxiong Wang, editors, ASIACRYPT 2020, Part II, volume 12492 of LNCS, pages 785–814. Springer, Cham, December 2020.

LLMP93. A. K. Lenstra, H. W. Lenstra, M. S. Manasse, and J. M. Pollard. The number field sieve. In Arjen K. Lenstra and Hendrik W. Lenstra, editors, The development of the number field sieve, pages 11–42, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

Pag02. Rasmus Pagh. Low redundancy in static dictionaries with constant query time. SIAM J. Comput., 31(2):353–363, February 2002.

Pol75. J. M. Pollard. A monte carlo method for factorization. BIT, 15(3):331–334, September 1975.

PQR22. Jiaxin Pan, Chen Qian, and Magnus Ringerud. Signed (group) Diffie-Hellman key exchange with tight security. Journal of Cryptology, 35(4):26, October 2022.

PRZ24. Jiaxin Pan, Doreen Riepel, and Runzhi Zeng. Key exchange with tight (full) forward secrecy via key confirmation. In Marc Joye and Gregor Leander, editors, EUROCRYPT 2024, Part VII, volume 14657 of LNCS, pages 59–89. Springer, Cham, May 2024.

PW22. Jiaxin Pan and Benedikt Wagner. Lattice-based signatures with tight adaptive corruptions and more. In Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe, editors, PKC 2022, Part II, volume 13178 of LNCS, pages 347–378. Springer, Cham, March 2022.

PWZ23a. Jiaxin Pan, Benedikt Wagner, and Runzhi Zeng. Lattice-based authenticated key exchange with tight security. In Helena Handschuh and Anna Lysyanskaya, editors, CRYPTO 2023, Part V, volume 14085 of LNCS, pages 616–647. Springer, Cham, August 2023.

PWZ23b. Jiaxin Pan, Benedikt Wagner, and Runzhi Zeng. Tighter security for generic authenticated key exchange in the QROM. In Jian Guo and Ron Steinfeld, editors, ASIACRYPT 2023, Part IV, volume 14441 of LNCS, pages 401–433. Springer, Singapore, December 2023.

SAB+20. Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

SXY18. Tsunekazu Saito, Keita Xagawa, and Takashi Yamakawa. Tightly-secure key-encapsulation mechanism in the quantum random oracle model. In Jesper Buus Nielsen and Vincent Rijmen, editors, EUROCRYPT 2018, Part III, volume 10822 of LNCS, pages 520–551. Springer, Cham, April / May 2018.

WMHT18. Yuyu Wang, Takahiro Matsuda, Goichiro Hanaoka, and Keisuke Tanaka. Memory lower bounds of reductions revisited. In Jesper Buus Nielsen and Vincent Rijmen, editors, EUROCRYPT 2018, Part I, volume 10820 of LNCS, pages 61–90. Springer, Cham, April / May 2018.

Xag24. Keita Xagawa. Signatures with memory-tight security in the quantum random oracle model. In Marc Joye and Gregor Leander, editors, EUROCRYPT 2024, Part VII, volume 14657 of LNCS, pages 30–58. Springer, Cham, May 2024.

Supporting Material

A Instantiations

We give two lattice-based instantiations of $\texttt{aSigKEM}$. The first instantiation is based on the lattice-based parameter-lossy encryption [PWZ23b] and the memory-tight signature scheme proposed in [Xag24]. They are based on hardness problems over lattices (LWE and SIS). This instantiation provides a tighter security bound. The second one combines the Kyber KEM [SAB+20] and the Dilithium signature scheme [LDK+20]. Both schemes are based on hardness problems over module lattices (Module-LWE and Module-SIS), and they are selected algorithms in the NIST PQC Standardization process.

To illustrate our results on memory-tightness of reductions, in this section, we only provide asymptotic bounds. Let ϵ_{LWE} and ϵ_{SIS} denote the computational bounds (i.e., the probabilities of breaking the assumptions) for the LWE and SIS problems, respectively. Similarly, let m^*_{LWE} and m^*_{SIS} represent the memory bounds required to break these problems. For a cryptographic scheme Π, we similarly write ϵ_{Π} and m^*_{Π} to denote the computational bounds of breaking Π and the reduced memory bounds required to break Π, respectively.

Instantiation based on Lattices. We use two lattice-based schemes to instantiate $\texttt{aSigKEM}$. The first one is the lattice-based parameter-lossy encryption, denoted as $\mathsf{PLE}_{\mathsf{LWE}}$ (cf. [PWZ23b, Figure 2]), which is proposed in [PWZ23b]. By [PWZ23b, Lemma 3.5, 3.6, and 3.7], we have

$$\epsilon_{\mathsf{PLE}_{\mathsf{LWE}}} \leq \mathrm{O}(\lambda) \cdot \epsilon_{\mathsf{LWE}} + \mathsf{negl}(\lambda).$$

Although the authors of [PWZ23b] do not study the memory-tightness of their reduction and the relation between parameter-lossy encryption and IND-CPA security, we find that: (1) The reduction given in [PWZ23b, Lemma 3.6] can easily be adapted to be memory-tight, since we only need to maintain the parameter, and all key pairs can be simulated by a (pseudo-)random function. (2) Parameter-lossy encryption tightly implies IND-CPA security in the multi-user, multi-challenge setting. Therefore, we can apply Theorem 3 to $\mathsf{PLE}_{\mathsf{LWE}}$ and get a lattice-based OW-PCA secure scheme (denoted as $\mathsf{KEM}_{\mathsf{LWE}}$) such that

$$\epsilon_{\mathsf{KEM}_{\mathsf{LWE}}} \leq \mathrm{O}(\lambda) \cdot \epsilon_{\mathsf{LWE}} + \mathsf{negl}(\lambda) \quad \text{and} \quad m^*_{\mathsf{KEM}_{\mathsf{LWE}}} \approx m^*_{\mathsf{LWE}}.$$

Then we discuss how to instantiate the signature part of $\texttt{aSigKEM}$. In [Xag24], Xagawa proposes a lattice-based signature, denoted as $\mathsf{SIG}_{\mathtt{Xag}}$, with a memory-tight reduction such that

$$\epsilon^{\mathsf{sEUF\text{-}1CMA}}_{\mathsf{SIG}_{\mathtt{Xag}}} \leq \mathrm{O}(1) \cdot \epsilon_{\mathsf{LWE}} + \epsilon_{\mathsf{SIS}} + \mathsf{negl}(\lambda) \quad \text{and} \quad m^*_{\mathsf{SIG}_{\mathtt{Xag}}} \approx m^*_{\mathsf{LWE}} \approx m^*_{\mathsf{SIS}}.$$

By applying Lemma 10, we have

$$\epsilon^{\mathsf{MU^c\text{-}sEUF\text{-}1CMA}}_{\mathsf{SIG}_{\mathtt{Xag}}} \leq \mathrm{O}(Q_{\mathrm{PK}}) \cdot (\epsilon_{\mathsf{LWE}} + \epsilon_{\mathsf{SIS}}) + \mathsf{negl}(\lambda) \quad \text{and} \quad m^*_{\mathsf{SIG}_{\mathtt{Xag}}} \approx m^*_{\mathsf{LWE}} \approx m^*_{\mathsf{SIS}},$$

where the ϵ here denotes the advantage against MU^c-sEUF-1CMA security (cf. Definition 3).

Finally, by instantiating $\texttt{aSigKEM}$ with $\mathsf{KEM}_{\mathsf{LWE}}$ and $\mathsf{SIG}_{\mathtt{Xag}}$ and applying Theorem 1, one can obtain a lattice-based AKE protocol (denoted as $\mathsf{AKE}_{\mathtt{lat}}$) such that

$$\begin{aligned}
&\epsilon_{\mathsf{AKE}_{\mathtt{lat}}} \leq \mathrm{O}(N_{\mathtt{user}} \cdot \lambda) \cdot (\epsilon_{\mathsf{LWE}} + \epsilon_{\mathsf{SIS}}) + \mathsf{negl}(\lambda) \\
&m^*_{\mathsf{AKE}_{\mathtt{lat}}} \approx m^*_{\mathsf{LWE}} \approx m^*_{\mathsf{SIS}},
\end{aligned}$$

where λ is the security parameter and Q_{PK} is the number of challenge public keys. Without loss of generality, we assume that $Q_{\mathrm{PK}} = N_{\mathtt{user}}$. Namely, the adversary does not issue repeated queries to the challenge public key oracle and the number of queries equals the number of users.

Instantiation from Module Lattices. We discuss how to use CRYSTALS-Kyber and CRYSTALS-Dilithium to instantiate $\texttt{aSigKEM}$. Note that in this instantiation, we view both CRYSTALS-Kyber and CRYSTALS-Dilithium as two secure underlying building blocks and do not consider their underlying bounds of MLWE and MSIS. Let $\epsilon_{\mathsf{kyber}}$ and m^*_{kyber} be the computational bound and required (reduced) memory for breaking the CPA PKE scheme in CRYSTALS-Kyber. Similarly, let $\epsilon_{\mathsf{dilit}}$ and m^*_{dilit} be the computational bound and required (reduced) memory for breaking CRYSTALS-Dilithium.

We start by considering the IND-CPA PKE scheme KYBER.CPAPKE. By [SAB+20, Section 4.3.1], KYBER.CPAPKE has IND-CPA security in the single-user, single-challenge setting. By using the generic construction in Figure 14 and applying Corollary 1, we transform KYBER.CPAPKE into a KEM scheme, denoted as $\mathsf{KEM}^{\mathsf{ow}}_{\mathsf{kyber}}$, that has OW-PCA security in the single-user, single-challenge setting. By applying Lemma 11, we lift its OW-PCA security to the multi-user, multi-challenge setting such that

$$\epsilon_{\mathsf{KEM}^{\mathsf{ow}}_{\mathsf{kyber}}} \leq \mathrm{O}((Q+1)(Q_{\mathrm{CT}}+1)) \cdot \epsilon_{\mathsf{kyber}} \quad \text{and} \quad m^*_{\mathsf{KEM}^{\mathsf{ow}}_{\mathsf{kyber}}} \approx m^*_{\mathsf{kyber}}.$$

where Q and Q_{CT} bound the total number of direct KEM oracle queries and the number of direct CHALCT queries, respectively, as in Lemma 11.

Then, we consider the signature scheme Dilithium in [LDK+20]. As discussed in [LDK+20, Section 6], Dilithium achieves the sEUF-CMA security in the single-user setting, which trivially implies the sEUF-1CMA security (cf. Definition 3). By applying Lemma 10, one can show that Dilithium achieves the MU^c-sEUF-1CMA security such that

$$\epsilon^{\mathsf{MU}^c\text{-}\mathsf{sEUF}\text{-}\mathsf{1CMA}}_{\mathsf{Dilithium}} \leq \mathrm{O}(Q_{\mathrm{PK}}) \cdot \epsilon_{\mathsf{dilit}} \quad \text{and} \quad m^*_{\mathsf{Sign}^{\mathsf{mu}}_{\mathsf{dilithium}}} \approx m^*_{\mathsf{dilit}}.$$

Finally, we use $\mathsf{KEM}^{\mathsf{ow}}_{\mathsf{kyber}}$ and $\mathsf{Sign}_{\mathsf{dilithium}}$ to instantiate $\mathtt{aSigKEM}$ and denote the resulting AKE protocol as $\mathsf{AKE}_{\mathtt{mlat}}$. By applying Theorem 1, we have

$$\begin{aligned} &\epsilon_{\mathsf{AKE}_{\mathtt{mlat}}} \leq \mathrm{O}((Q+1)(Q_{\mathrm{CT}}+1)) \cdot \epsilon_{\mathsf{kyber}} + \mathrm{O}(N_{\mathtt{user}}) \cdot \epsilon_{\mathsf{dilit}}, \\ &m^*_{\mathsf{AKE}_{\mathtt{mlat}}} \approx m^*_{\mathsf{kyber}}, \quad \text{and} \quad m^*_{\mathsf{AKE}_{\mathtt{mlat}}} \approx m^*_{\mathsf{dilit}}. \end{aligned} \tag{1}$$

Here Q and Q_{CT} refer to the KEM adversary constructed in Theorem 1. Although our MLWE-based AKE protocol does not have a fully tight reduction, its memory-tightness only has a loss factor proportional to the number of users (and the size of the security parameter) and thus has a better memory-efficient reduction than the previous AKE constructions that are based on module lattices (e.g., [FSXY12,FSXY13]).

Remark 1 (On the multi-user, multi-challenge security of Kyber*).* The security loss $(Q+1)(Q_{\mathrm{CT}}+1)$ in Equation (1) originates from the general transformation from single-user, single-challenge security to multi-user, multi-challenge security (cf. Lemma 11). To improve the security loss, a possible way is to directly analyze the security of the CPA PKE scheme of Kyber in the multi-user, multi-challenge setting, which may give a tighter bound than simply applying the general transformation. Note that this will improve the security loss for our AKE protocol, and also give a better security bound for all multi-user, multi-challenge applications based on Kyber (e.g., post-quantum TLS, KEM-TLS, etc.). We leave this as an open question for future work.

B More Detailed Explanations of Our New AKE Model

In this section, we discuss several motivations behind the definitions of our AKE model ind-wFS-KCI. In Supp. Mat. C, we also compare our model with prior models.

Repeated Queries to Pk. When $\mathcal{A}$ queries $\textsc{Pk}(u \in \mathcal{U})$, the oracle returns pk_u. If $\mathcal{A}$ queries Pk repeatedly on the same user identifier, then the same public key is returned. This method of modelling public key generation is proposed by [GGJT22,JK22]. It allows security reductions to simulate all public keys in a memory-tight manner.

On the Definition of Session Identifiers. We first give a brief review on two previous definitions of session identifiers in [FSXY12,JKRS21,PWZ23a]. In [FSXY12], session identifiers are defined as protocol messages. For instance, if u sent M_u to v and received M_v, then this session is identified as $(u, v, \mathsf{M}_u, \mathsf{M}_v)$. In [JKRS21,PWZ23a], session identifiers are defined as an internal counter controlled by the simulator. Every query to oracles $\textsc{Init}_\textsc{I}$ and $\textsc{Der}_\textsc{R}$ will increment this counter and use it as a session identifier. This counter can be viewed as a sequence number, with each session assigned a unique value. In Remark 4, we discuss why this approach is unlikely to be memory-tight.

Our definition of session identifiers is a combination of these two approaches, namely, it consists of counter numbers and the protocol messages. For an initiator session, its identifier is of the format

$$\mathsf{sID} := (\mathsf{seq}_1, \mathsf{seq}_2, u, v, \mathsf{M}_u, \mathsf{M}_v), \qquad \text{(Initiator session)}$$

where seq_1 and seq_2 are the initial and final sequence numbers, u and v are identities of initiator and responder, and M_u and M_v are protocol transcripts. Correspondingly, the identifier of a responder session is defined as

$$\mathsf{sID} := (\mathsf{seq}, u, v, \mathsf{M}_u, \mathsf{M}_v), \qquad \text{(Responder session)}$$

where seq is the sequence number. Since a responder session produces a session key when it receives the protocol message, here we just need one sequence number to define the session identifier of a responder session.

Now we explain why our definition includes both sequence numbers and protocol messages:

- On the one hand, if we only use sequence numbers (more concretely, a reduction-controlled counter or random number) to define session identifiers (as [JKRS21,PWZ23a] did), then we may have to record session keys to simulate Rev and Test. This is because such sequence numbers cannot record the received messages of a session without using lists if such messages are generated by the adversary and not reduction-controlled.
- On the other hand, if we only use protocol messages to define session identifiers (as [FSXY12] did), then we may have to remember all established session identifiers. This is because the adversary can use protocol messages to forge a session identifier.

For example, the initiator session identifiers in our model include two sequence numbers. The first one, seq_1, is used to identify an incomplete initiator session (since there is no responder protocol message yet). The second one, seq_2, is used to identify the complete session along with seq_1. Such a definition helps us to exclude the case that the adversary tries querying any initiator session that does not exist. For example, if we only use seq_1 to identify this session, then we have to remember the history of whether such a session has been finished. Note that our definition requires that if the adversary sends repeating queries to $\textsc{Der}_\textsc{I}$, then the oracle outputs the same seq_2.

Furthermore, including protocol messages in the session identifier makes it possible to recover a session key in a memory-tight way without recording. Intuitively, this definition forces the adversary to remember all protocol messages, effectively shifting the memory usage of recording these messages onto the adversary.

Modelling attacks and security. Oracles $\textsc{Init}_\textsc{I}, \textsc{Der}_\textsc{R}$, and $\textsc{Der}_\textsc{I}$ are used to model replay attacks and man-in-the-middle (MiM) attacks by allowing adversaries to fully control the transmission of all protocol messages. We also allow the adversary to reveal any established session key by querying the oracle Rev.

To capture key-compromise impersonation (KCI) attacks and weak forward-secrecy (wFS), we additionally allow an adversary to corrupt the long-term secret key of any user by querying Cor. Roughly speaking, to launch a KCI attack, $\mathcal{A}$ firstly corrupts a target user and then impersonates another user to

this target user. To launch a wFS attack against a passive session, an adversary can honestly forward all protocol messages of this session and then corrupt the long-term secret keys of this session's participants. All corrupted users are recorded in the set $\mathcal{S}_{\mathtt{cor}}$. Together with oracles $\textsc{Init}_{\mathrm{I}}, \textsc{Der}_{\mathrm{R}}$, and $\textsc{Der}_{\mathrm{I}}$, an adversary can easily perform KCI or wFS attacks.

We define *freshness* and *validity* to exclude all trivial attacks:

- Revealing a session key via Rev makes the attack against this session trivial. All sessions revealed by querying Rev are recorded in the set $\mathcal{S}_{\mathtt{rev}}$. Moreover, if a session is revealed, then all matching sessions of this session are considered revealed (cf. Definition 6). In this model, we use an auxiliary procedure Fresh to check whether a session is fresh.
- For a responder session $\mathsf{sID} = (\mathsf{seq}, u, v, \mathsf{M}_u, \mathsf{M}_v)$, if the session initiator u is corrupted and (u, M_u) is not generated from $\textsc{Init}_{\mathrm{I}}$, then we consider that this session is trivially broken by $\mathcal{A}$. This is because, if $\mathcal{A}$ corrupted u and sends its protocol messages impersonating u, then there is no secret information in this session for security.
- For an initiator session $\mathsf{sID} = (\mathsf{seq}_1, \mathsf{seq}_2, u, v, \mathsf{M}_u, \mathsf{M}_v)$, if the session responder v is corrupted and $(u, v, \mathsf{M}_u, \mathsf{M}_v)$ is not in $\mathcal{S}_{\mathtt{msg}}$ (which means that sID does not have a matching session), then we consider that this session is trivially broken.

The models in [JKRS21,PWZ23a,PWZ23b] use a table to list all valid attacks. Here, we use an auxiliary procedure Valid to check whether a session is valid. In the security proof, this allows us to be precise about the memory complexity of the reduction and to exclude unnecessary use of memory. The formal definitions of freshness and validity are given below, and the procedures for determining them are shown in Figure 4. A more detailed comparison with prior models can be found in Supp. Mat. C.

Remark 2 (Multiple (partially) matching sessions). In our model, if an adversary can construct a session with more than one matching session, then we consider such a session insecure. This allows us to exclude those protocols that can establish more than one matching session (which is not supposed to happen for a "reasonable" AKE protocol). Such an approach of defining AKE security is also used in [JKRS21,PWZ23a,PRZ24].

Different from some previous AKE models (e.g., the model in [PWZ23b]), our model does not consider multiple "partially" matching sessions. Here partially matching means that (for two-message AKE) a responder session has more than one initiator session that has the same initiator message as this responder session. While prior models explicitly capture this for clearer clarification, we leave it implicit to simplify the description of the model.

For a two-message AKE protocol, if the initiator's first message does not contain sufficient uncertainty (e.g., sufficiently high min-entropy), then it fails to carry enough fresh, session-specific secret information. Consequently, an adversary may be able to predict the message, together with the associated ephemeral secret, and use this information to impersonate the initiator session. Therefore, a secure two-message AKE protocol in the public-key setting implicitly requires the first protocol message to have sufficiently high uncertainty, so that the probability of multiple partially matching sessions remains negligible.

Remark 3 (Reuse incomplete initiator sessions). Different from the previous models, our AKE model allows the adversary to reuse incomplete initiator sessions. For example, the adversary can first query $\textsc{Init}_{\mathrm{I}}$ to get a session sequence number seq_1, and then query $\textsc{Der}_{\mathrm{I}}$ on the same seq_1 but different responder messages. The oracle $\textsc{Der}_{\mathrm{I}}$ does not check whether the seq_1 has been used. Since our model uses protocol messages to define sessions, as long as the adversary sends different responder messages, these queries (with the same seq_1 but different M_v) still produce different sessions. If the adversary issues repeated queries to $\textsc{Der}_{\mathrm{I}}$, by lines 33 to 38, the oracle will respond with the same session identity and produce the same session key.

We make this modification for the following reasons:

- If we record whether an incomplete initiator session (i.e., seq_1) has been used, then the memory-usage of this information is linear in the number of sessions (which is very large). From the perspective of constructing a reduction, this memory usage seems unavoidable, because an adversary can always activate an initiator session, and the reduction has to keep the information since it does not know if the adversary will finish it later. The adversary can leave the session incomplete and force the reduction to store it during the game.
- To address this, we allow the session initiator to always accept new responder messages, regardless of whether the session sequence number seq_1 has been used. This, on one hand, is a stronger property.

For example, in our protocol, the KEM scheme should be OW-PCA secure, not OW-PCA secure with one Pco query (and similarly, the IND-one-time-CCA security is also not sufficient). On the other hand, we may understand this in this way: To achieve a memory-tight reduction, the AKE protocol itself should have the ability to resist such reuse-incomplete-session attacks and our new model captures this property. If a protocol does not provide security against these attacks, then one may conclude that the memory of recording sessions provides an advantage to help the adversary break the underlying building block of the protocol.

The technical details can be found in the proof of Theorem 2 and Remark 4.

C Relation to the Model of Pan et al.

We compare our model ind-wFS-KCI with a single-Test model adapted from the two-message AKE model of Pan, Wagner, and Zeng [PWZ23b]. Their model captures man-in-the-middle and replay attacks, weak forward secrecy, and key-compromise impersonation, without session-state or ephemeral-secret reveal. In particular, its adversary may issue at most one Test query. We retain this restriction in the comparison model, denoted by ind-wFS-KCI′, while ind-wFS-KCI permits multiple Test queries.

We express the comparison model using our notation and our convention for the distinguishing advantage. Its freshness and validity conditions are adapted to the treatment of matching sessions described below; the comparison is with this adapted game. Each session identifier sID is a positive integer. Besides the lists and variables in Section 3.2, we record the following information:

- $\texttt{Type}[\mathsf{sID}]$ stores the type of the session. If sID is an initiator session (resp., responder session), then $\texttt{Type}[\mathsf{sID}] = \texttt{I}$ (resp., $\texttt{Type}[\mathsf{sID}] = \texttt{R}$).
- $\texttt{Init}[\mathsf{sID}]$ stores the user identifier of the session initiator of sID.
- $\texttt{Imsg}[\mathsf{sID}]$ stores the first protocol message (initiator message) of sID.
- $\texttt{Resp}[\mathsf{sID}]$ stores the user identifier of the session responder of sID.
- $\texttt{Rmsg}[\mathsf{sID}]$ stores the final protocol message (responder message) of sID.

The adapted single-Test game and its formal definition are presented in Figure 10 and Definition 8.

Syntax. The two games differ in the following ways:

1. In ind-wFS-KCI′, a session identifier is a positive integer produced by a counter cnt which increments every time a session is produced, rather than the combination of random sequence numbers and protocol messages.
2. Since ind-wFS-KCI′ does not use random sequence numbers and protocol messages to define session identifiers, we need the variables defined before to record necessary session information.
3. ind-wFS-KCI′ does not allow initiator session reuse, while ind-wFS-KCI permits it. This means ind-wFS-KCI′ is *conceptually weaker* than ind-wFS-KCI.

Test Queries. The comparison model permits at most one Test query in total, including a query that returns $\perp$. Thus its adversary cannot test both endpoints of an exchange. The oracle returns the real session key when $b = 0$ and a fresh uniform key when $b = 1$. It needs neither a cache for repeated tests nor a rule for sharing random test keys across matching sessions.

Freshness and Validity. We adapt the attack conditions of [PWZ23b, Table 1] using the message records and the Fresh and Valid procedures in Figure 10. As discussed in Remark 2, our treatment does not include the separate condition for multiple partially matching sessions in that work. The comparison below concerns the explicitly defined adapted game ind-wFS-KCI′. In particular, the single-Test restriction does not remove freshness checks: revealing the tested session or its matching session must still be excluded for an ordinary honest exchange.

Definition 8. *Let* $\Pi := (\mathsf{Setup}, \mathsf{KG}, \mathsf{Init}_{\mathrm{I}}, \mathsf{Der}_{\mathrm{R}}, \mathsf{Der}_{\mathrm{I}})$ *be an AKE protocol and consider the AKE game* ind-wFS-KCI′ *defined in Figure 10. We say* AKE *is secure under the model* ind-wFS-KCI′ *if, for all PPT adversaries* $\mathcal{A}$ *making at most one* Test *query, the following advantage is negligible in the security parameter* λ*:*

$$\mathbf{Adv}_{\Pi}^{\text{ind-wFS-KCI}'}(\mathcal{A}) := \left| \Pr\left[\text{ind-wFS-KCI}'^{\mathcal{A}}_{\Pi} \Rightarrow 1 \right] - 1/2 \right|$$

Now we are ready to show that ind-wFS-KCI is *not weaker* than ind-wFS-KCI′, namely, if an AKE protocol Π is secure in the ind-wFS-KCI model, then it is also secure in the ind-wFS-KCI′ model. We prove this via contradiction: For any adversary $\mathcal{A}'$, if $\mathcal{A}'$ breaks Π in the ind-wFS-KCI′ model, then we can construct another adversary to break Π in the ind-wFS-KCI model. The formal statement is given in Theorem 2.

Theorem 2 (ind-wFS-KCI $\implies$ ind-wFS-KCI′)**.** *Let* Π *be a two-message AKE protocol. For any adversary* $\mathcal{A}'$ *that breaks the* ind-wFS-KCI′ *security of* Π*, there exists an adversary* $\mathcal{A}$ *that breaks the* ind-wFS-KCI *security of* Π*.*

Game ind-wFS-KCI$'^{\mathcal{A}}_{\Pi}$

00 $\mathsf{cnt} := 0$
01 $\mathsf{testUsed} := \mathbf{false}$
02 $b \leftarrow \{0,1\}$, $\mathsf{par} \leftarrow \mathsf{Setup}$
03 $O_1 := (\textsc{Init}_{\mathrm{I}}, \textsc{Der}_{\mathrm{I}}, \textsc{Der}_{\mathrm{R}})$
04 $O_2 := (\textsc{Pk}, \textsc{Cor}, \textsc{Rev}, \textsc{Test})$
05 $b' \leftarrow \mathcal{A}^{O_1,O_2}(\mathsf{par})$
06 **for** $\mathsf{sID}^* \in \mathcal{S}_{\mathtt{test}}$:
07 **if** $\mathsf{Fresh}(\mathsf{sID}^*) = 0 \ \vee \ \mathsf{Valid}(\mathsf{sID}^*) = 0$
08 **return** $b' \leftarrow \{0,1\}$
09 **return** $[\![b' = b]\!]$

Oracle $\textsc{Init}_{\mathrm{I}}(u, v)$

10 $\mathsf{cnt} := \mathsf{cnt} + 1$, $\mathsf{sID} := \mathsf{cnt}$
11 $(\mathtt{Init}[\mathsf{sID}], \mathtt{Resp}[\mathsf{sID}]) := (u, v)$
12 $(\mathsf{M}_u, \mathsf{st}) \leftarrow \mathsf{Init}_{\mathrm{I}}(u, v, \mathsf{pk}_u, \mathsf{sk}_u, \mathsf{pk}_v)$
13 $\mathtt{Type}[\mathsf{sID}] := \mathtt{I}$, $\mathtt{Imsg}[\mathsf{sID}] := \mathsf{M}_u$
14 $\mathcal{L}_{\mathtt{st}}[\mathsf{sID}] := (u, v, \mathsf{M}_u, \mathsf{st})$
15 $\mathcal{S}_{\mathtt{msg}} := \mathcal{S}_{\mathtt{msg}} \cup \{(u, \mathsf{M}_u)\}$
16 **return** $(\mathsf{sID}, \mathsf{M}_u)$

Oracle $\textsc{Der}_{\mathrm{R}}(u, v, \mathsf{M}_u)$

17 $\mathsf{cnt} := \mathsf{cnt} + 1$, $\mathsf{sID} := \mathsf{cnt}$
18 $(\mathtt{Init}[\mathsf{sID}], \mathtt{Resp}[\mathsf{sID}]) := (u, v)$
19 $(\mathsf{M}_v, \mathsf{K}) \leftarrow \mathsf{Der}_{\mathrm{R}}(u, v, \mathsf{pk}_v, \mathsf{sk}_v, \mathsf{pk}_u, \mathsf{M}_u)$
20 **if** $\mathsf{K} = \bot$: **return** $\bot$
21 **if** $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$
22 $\mathcal{S}_{\mathtt{msg}} := \mathcal{S}_{\mathtt{msg}} \cup \{(u, v, \mathsf{M}_u, \mathsf{M}_v)\}$
23 $\mathtt{Type}[\mathsf{sID}] := \mathtt{R}$
24 $\mathtt{Imsg}[\mathsf{sID}] := \mathsf{M}_u$, $\mathtt{Rmsg}[\mathsf{sID}] := \mathsf{M}_v$
25 $\mathcal{S}_{\mathtt{sid}} := \mathcal{S}_{\mathtt{sid}} \cup \{\mathsf{sID}\}$
26 $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}] := \mathsf{K}$
27 **return** $(\mathsf{sID}, \mathsf{M}_v)$

Oracle $\textsc{Der}_{\mathrm{I}}(\mathsf{sID}, \mathsf{M}_v)$

28 **if** $\mathsf{sID} \in \mathcal{S}_{\mathtt{sid}}$ // No session reuse
29 **return** $\bot$
30 $\mathcal{S}_{\mathtt{sid}} := \mathcal{S}_{\mathtt{sid}} \cup \{\mathsf{sID}\}$
31 $(u, v, \mathsf{M}_u, \mathsf{st}) := \mathcal{L}_{\mathtt{st}}[\mathsf{sID}]$
32 **if** $\mathsf{st} = \bot$: **return** $\bot$
33 $\mathsf{K} := \mathsf{Der}_{\mathrm{I}}(u, v, \mathsf{pk}_u, \mathsf{sk}_u, \mathsf{pk}_v, \mathsf{st}, \mathsf{M}_v)$
34 **if** $\mathsf{K} = \bot$: **return** $\bot$
35 $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}] := \mathsf{K}$, $\mathtt{Rmsg}[\mathsf{sID}] := \mathsf{M}_v$
36 **return** sID

Oracle $\textsc{Pk}(u \in \mathcal{U})$

37 **if** $\mathcal{L}_{\mathtt{key}}[u] = \bot$
38 $\mathcal{L}_{\mathtt{key}}[u] := (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}$
39 **return** pk

Oracle $\textsc{Cor}(u)$

40 $\mathcal{S}_{\mathtt{cor}} := \mathcal{S}_{\mathtt{cor}} \cup \{u\}$
41 $(\mathsf{pk}_u, \mathsf{sk}_u) := \mathcal{L}_{\mathtt{key}}[u]$
42 **return** sk_u

Oracle $\textsc{Rev}(\mathsf{sID})$

43 **if** $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}] = \bot$: **return** $\bot$
44 $\mathcal{S}_{\mathtt{rev}} := \mathcal{S}_{\mathtt{rev}} \cup \{\mathsf{sID}\}$
45 **return** $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}]$

Oracle $\textsc{Test}(\mathsf{sID})$

46 **if** testUsed: **return** $\bot$
47 $\mathsf{testUsed} := \mathbf{true}$
48 **if** $\mathcal{L}_{\mathtt{SK}}[\mathsf{sID}] = \bot$: **return** $\bot$
49 $\mathcal{S}_{\mathtt{test}} := \mathcal{S}_{\mathtt{test}} \cup \{\mathsf{sID}\}$
50 $\mathsf{SK}_0 := \mathcal{L}_{\mathtt{SK}}[\mathsf{sID}]$, $\mathsf{SK}_1 \leftarrow \mathcal{SK}$
51 **return** SK_b

Alg ComputeMS(sID)

52 $(u, v) := (\mathtt{Init}[\mathsf{sID}], \mathtt{Resp}[\mathsf{sID}])$
53 $(\mathsf{M}_u, \mathsf{M}_v) := (\mathtt{Imsg}[\mathsf{sID}], \mathtt{Rmsg}[\mathsf{sID}])$
54 $\mathfrak{M}(\mathsf{sID}) := \emptyset$
55 **for** $\mathsf{sID}' \in \mathcal{S}_{\mathtt{sid}} \backslash \{\mathsf{sID}\}$
56 $(u', v') := (\mathtt{Init}[\mathsf{sID}'], \mathtt{Resp}[\mathsf{sID}'])$
57 $(\mathsf{M}'_{u'}, \mathsf{M}'_{v'}) := (\mathtt{Imsg}[\mathsf{sID}'], \mathtt{Rmsg}[\mathsf{sID}'])$
58 **if** $\mathtt{Type}[\mathsf{sID}] \neq \mathtt{Type}[\mathsf{sID}']$
59 $\wedge \ (u, v, \mathsf{M}_u, \mathsf{M}_v) = (u', v', \mathsf{M}'_{u'}, \mathsf{M}'_{v'})$
60 $\mathfrak{M}(\mathsf{sID}) := \mathfrak{M}(\mathsf{sID}) \cup \{\mathsf{sID}'\}$
61 **return** $\mathfrak{M}(\mathsf{sID})$

Alg Fresh(sID)

62 $\mathfrak{M}(\mathsf{sID}) := \mathsf{ComputeMS}(\mathsf{sID})$
63 **if** $|\mathfrak{M}(\mathsf{sID})| > 1$: **return** 1
64 **if** $\mathsf{sID} \in \mathcal{S}_{\mathtt{rev}}$: **return** 0
65 **for** $\mathsf{sID}' \in \mathfrak{M}(\mathsf{sID})$:
66 **if** $\mathsf{sID}' \in \mathcal{S}_{\mathtt{rev}}$: **return** 0
67 **return** 1

Alg Valid(sID)

68 $\mathfrak{M}(\mathsf{sID}) := \mathsf{ComputeMS}(\mathsf{sID})$
69 **if** $|\mathfrak{M}(\mathsf{sID})| > 1$: **return** 1
70 $(u, v) := (\mathtt{Init}[\mathsf{sID}], \mathtt{Resp}[\mathsf{sID}])$
71 $(\mathsf{M}_u, \mathsf{M}_v) := (\mathtt{Imsg}[\mathsf{sID}], \mathtt{Rmsg}[\mathsf{sID}])$
72 **if** $\mathtt{Type}[\mathsf{sID}] = \mathtt{R}$ // Responder session
73 **if** $u \notin \mathcal{S}_{\mathtt{cor}} \vee (u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$
74 **return** 1
75 **if** $\mathtt{Type}[\mathsf{sID}] = \mathtt{I}$ // Initiator session
76 **if** $v \notin \mathcal{S}_{\mathtt{cor}} \ \vee \ (u, v, \mathsf{M}_u, \mathsf{M}_v) \in \mathcal{S}_{\mathtt{msg}}$
77 **return** 1
78 **return** 0

Fig. 10. Single-Test comparison game ind-wFS-KCI$'$ in Definition 8, adapted from [PWZ23b]. The adversary issues at most one Test query. In ind-wFS-KCI$'$, $\mathcal{A}$ queries $\textsc{Pk}(u)$ to access the public key of user u, and all oracles access the key pair of user u via querying Pk first and then finding the key pair from $\mathcal{L}_{\mathtt{key}}[u]$. This generates keys on demand instead of providing all public keys initially as in [PWZ23b].

More specifically, if $\mathcal{A}'$ issues $Q_{\textsc{Pk}}, Q_{\textsc{Rev}}, Q_{\textsc{Cor}}, Q_{\textsc{Test}}$, $Q_{\mathtt{H}}$, and $Q_{\mathtt{send}}$ queries to the oracles Pk *(including direct and indirect queries, i.e., queries from reductions),* Rev, Cor, Test, H, *and* $(\textsc{Init}_{\mathrm{I}}, \textsc{Der}_{\mathrm{I}}, \textsc{Der}_{\mathrm{R}})$ *in* ind-wFS-KCI$'$*, respectively, with $Q_{\textsc{Test}} \leq 1$, then there exists an adversary $\mathcal{A}$ such that*

$$\mathbf{Adv}^{\text{ind-wFS-KCI}}_{\Pi}(\mathcal{A}) = \mathbf{Adv}^{\text{ind-wFS-KCI}'}_{\Pi}(\mathcal{A}'),$$
$$\mathbf{T}(\mathcal{A}) \approx \mathbf{T}(\mathcal{A}'),$$

where $\mathcal{A}$ issues at most $Q_{\textsc{Pk}}, Q_{\textsc{Rev}}, Q_{\textsc{Cor}}, Q_{\textsc{Test}}$, and $Q_{\mathtt{send}}$ queries to the oracles Pk, Rev, Cor, Test, *and* $(\textsc{Init}_{\mathrm{I}}, \textsc{Der}_{\mathrm{I}}, \textsc{Der}_{\mathrm{R}})$ *in* ind-wFS-KCI*, respectively.*

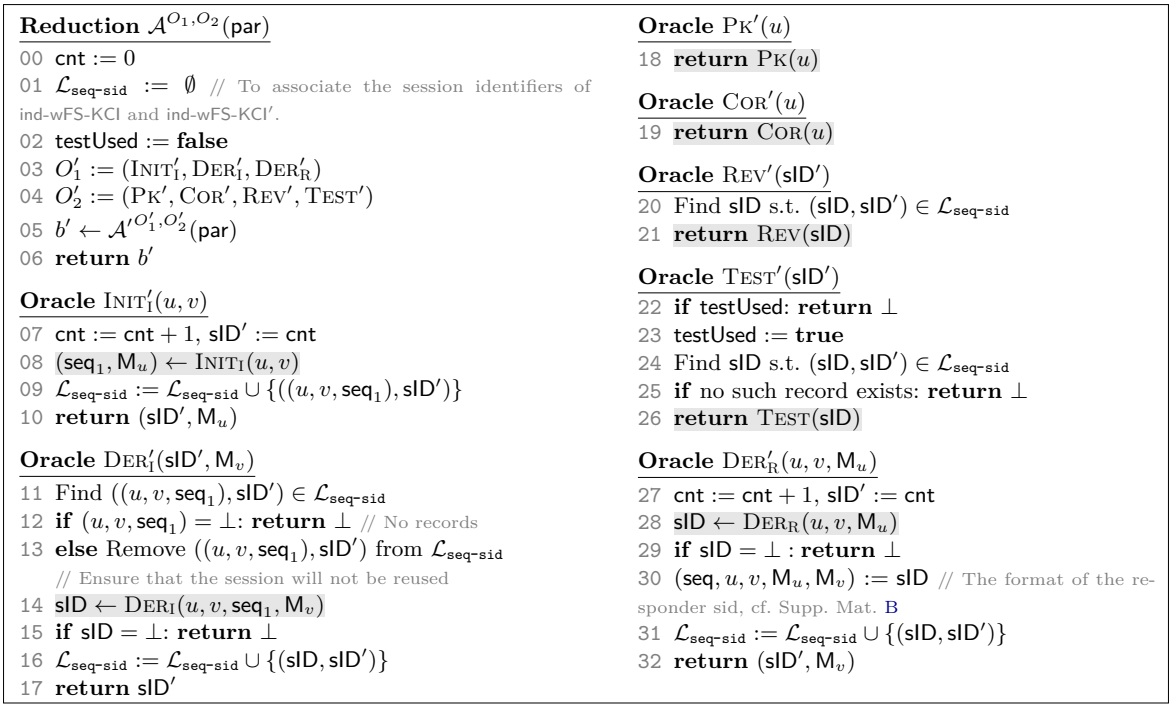

Fig. 11. Reduction $\mathcal{A}$ for proving Theorem 2. $\mathcal{A}$ plays ind-wFS-KCI and simulates ind-wFS-KCI′ for the adversary $\mathcal{A}'$. It has access to oracles $O_1 := (\textsc{Init}_{\mathrm{I}}, \textsc{Der}_{\mathrm{I}}, \textsc{Der}_{\mathrm{R}})$ and $O_2 := (\textsc{Pk}, \textsc{Cor}, \textsc{Rev}, \textsc{Test})$. Since the oracles of ind-wFS-KCI′ share the same names with ind-wFS-KCI, we use " ′ " to mark all oracles in ind-wFS-KCI′, and use sID' to distinguish from the session identifiers sID in ind-wFS-KCI. The highlighted code presents how $\mathcal{A}$ uses the oracles from ind-wFS-KCI to simulate ind-wFS-KCI′.

Theorem 2 states the relation between ind-wFS-KCI and ind-wFS-KCI′ in terms of time complexity and winning advantage. The reduction in Theorem 2 is not memory-tight (and is also unlikely to be memory-tight, as explained in Remark 4), but the goal of Theorem 2 is to show ind-wFS-KCI captures the security goals in ind-wFS-KCI′ and thus the security established by our model is not weaker than the adapted comparison model.

Proof (Theorem 2). We construct a straightforward adversary $\mathcal{A}$ that simply forwards all queries from $\mathcal{A}'$ to the ind-wFS-KCI game (via adapting the format of session identifiers). The construction of $\mathcal{A}$ is presented in Figure 11. It simulates ind-wFS-KCI′ in the following way:

- $\mathcal{A}$ uses a counter cnt to simulate the session identifiers sID' in ind-wFS-KCI′ and uses an internal (i.e., not accessible by $\mathcal{A}'$) list $\mathcal{L}_{\mathtt{seq\text{-}sid}}$ to maintain the session identifiers sID' in ind-wFS-KCI′ and the session identifiers sID in ind-wFS-KCI. When $\mathcal{A}'$ issues a query about sID', $\mathcal{A}$ finds the corresponding sID from $\mathcal{L}_{\mathtt{seq\text{-}sid}}$ and forwards the query to ind-wFS-KCI.
- The freshness and validity definitions of ind-wFS-KCI′ are equivalent to those in ind-wFS-KCI, so $\mathcal{A}$ only needs to find the corresponding session identifiers and forward all queries to ind-wFS-KCI. Since all session-related information (protocol messages, revealed sessions, corrupted parties, etc.) has been handled in ind-wFS-KCI, when simulating ind-wFS-KCI′, $\mathcal{A}$ does not need to store this information.

It is straightforward to show that $\mathcal{A}$ perfectly simulates ind-wFS-KCI′ for $\mathcal{A}'$, so the challenge bit used in ind-wFS-KCI is also the one (implicitly) used in ind-wFS-KCI′. Therefore, we have

$$\mathbf{Adv}_{\Pi}^{\text{ind-wFS-KCI}}(\mathcal{A}) = \mathbf{Adv}_{\Pi}^{\text{ind-wFS-KCI}'}(\mathcal{A}') \text{ and } \mathbf{T}(\mathcal{A}) \approx \mathbf{T}(\mathcal{A}'),$$

where the equation on time complexity is from the fact that $\mathcal{A}$ can search sID in $\mathcal{L}_{\mathtt{seq\text{-}sid}}$ via some efficient search algorithms (e.g., binary search in an ordered list using sID' as an index). This concludes the proof of Theorem 2.

Remark 4 (Memory cost of the comparison model). The proof of Theorem 2 also reveals the intuition on why it is hard to construct memory-tight reductions under the ind-wFS-KCI′ model. In ind-wFS-KCI′,

the session identifiers are defined as successive positive numbers. Since this structure is known, the AKE adversary can easily predict the session identifiers, so we cannot embed session information into them (and then use the technique in Section 4). This is why ind-wFS-KCI uses a random sequence number as a part of a session identifier.

Moreover, in ind-wFS-KCI′, the reduction needs to check whether an initiator session has been used, which is also essentially a dictionary problem discussed at the end of Section 1. This is why our new model ind-wFS-KCI allows initiator session reuse, otherwise it will incur an inherent loss $\Omega(N_{\mathtt{ses}})$ in memory.

Remark 5 (Extension to multiple test sessions). We can also consider a multi-TEST version of the adapted model of Pan, Wagner, and Zeng [PWZ23b]. More specifically, we remove the single-query restriction in ind-wFS-KCI′ and use one challenge bit b for all TEST queries. For this extension, we remove the testUsed check, and add a consistency check that, if the challenge bit b is set to 1, then the test queries on matching sessions must return the same random key. To achieve this, one can maintain a list that records $(\mathsf{sID}, \mathsf{sID}')$ if $\mathsf{sID}' \in \mathsf{ComputeMS}(\mathsf{sID})$ when sID is completed. Since the session-identifier translation preserves the users and protocol meesages, the way of maintaining session identifiers in the reduction in Theorem 2 can be easily extended to this multi-TEST version of ind-wFS-KCI′.

D Signatures with Memory-tight Reductions

Here we show that sEUF-1CMA implies MU^c-sEUF-1CMA. Our reduction is not TA-tight, but preserves the memory tightness.

Lemma 10 (sEUF-1CMA implies MU^c-sEUF-1CMA, memory-tightly). *Let* $\mathsf{SIG} = (\mathsf{Setup}, \mathsf{KG}, \mathsf{Sign}, \mathsf{Vrfy})$ *be a signature scheme where the randomness spaces of* KG *and* Sign *are* $\mathcal{R}_{\mathtt{k}}$ *and* $\mathcal{R}_{\mathtt{s}}$*, respectively. For any* $\mathcal{A}$ *against* MU^c-sEUF-1CMA *security of* SIG*, let* $N' \geq 1$ *be a fixed, a priori upper bound on the number of distinct user identifiers touched by any of its oracle queries, including indirect public-key requests. If* $\mathcal{A}$ *issues at most* Q_{PK} *queries to* PK *(including indirect requests), then there exists an* $\mathcal{F}(\mathcal{U}, \mathcal{R}_{\mathtt{k}}) \times \mathcal{F}(\mathcal{U} \times \mathcal{M}, \mathcal{R}_{\mathtt{s}}) \times \mathcal{F}(\mathcal{U}, [N'])$*-adversary* $\mathcal{B}$ *s.t.*

$$\begin{aligned}
&\mathbf{Adv}^{\mathsf{MU}^c\text{-}\mathsf{sEUF\text{-}1CMA}}_{\mathsf{SIG}}(\mathcal{A}) \leq e \cdot N' \cdot \mathbf{Adv}^{\mathsf{sEUF\text{-}1CMA}}_{\mathsf{SIG}}(\mathcal{B}),\\
&\mathbf{T}^*(\mathcal{B}) = \mathbf{T}(\mathcal{A}) + \mathrm{O}(Q_{\mathrm{PK}} + Q_{\mathtt{forge}} + Q_{\mathtt{sign}} + Q_{\mathtt{cor}}) \cdot \mathbf{T}(\mathsf{SIG}),\\
&\mathbf{Mem}^*(\mathcal{B}) = \mathbf{Mem}(\mathcal{A}) + \mathbf{Mem}(\mathsf{SIG}) + \log|\mathcal{U}| + \log N' + 1,
\end{aligned}$$

where $Q_{\mathrm{PK}}, Q_{\mathtt{forge}}, Q_{\mathtt{sign}}$*, and* $Q_{\mathtt{cor}}$ *are fixed upper bounds on the numbers of queries made by* $\mathcal{A}$ *to* PK *(including direct and indirect queries),* FORGE, SIGN*, and* COR*, and* $\mathcal{B}$ *makes at most* $Q_{\mathtt{sign}}$ *queries to* SIGN′ *and at most* $Q_{\mathtt{forge}}$ *queries to* FORGE′*. One may choose* $N' := \max\{1, \min\{|\mathcal{U}|, Q_{\mathrm{PK}}\}\}$*. This parameter is chosen before the execution, not from the realized transcript.*

Proof. We construct a reduction $\mathcal{B}$ in Figure 12. In the construction of $\mathcal{B}$, we use two random functions f_1 and f_2 to generate the randomnesses for KG and Sign, respectively. Here we use the memory-tight guessing technique proposed in [ACFK17, Section 3.4]. Fix N' as in the lemma (repeated calls to GetKey for one user do not count as distinct users). To randomly embed the challenge public key pk^*, we use a random function $f_3 : \mathcal{U} \to [N']$ and if $f_3(u^*) = 1$, then we let the public key of u^* be pk^*. Note that repeated (regardless of direct or indirect) queries to PK with the same user identifiers always produce the same key pairs.

Suppose that $\mathcal{A}$'s first successful FORGE query in the MU^c-sEUF-1CMA game is (u^*, m^*, σ^*). By the winning condition, $u^* \notin \mathcal{S}_{\mathtt{cor}}$. To embed pk^* into user u^*, we need $f_3(u^*) = 1$. However, we also require $f_3(u') \neq 1$ for every other user u' involved up to this query. For any fixed sequence of queries and answers up to this forgery in the original game, f_3 is chosen independently. Since at most N' distinct users are involved, these conditions hold with probability at least

$$\frac{1}{N'}\left(1 - \frac{1}{N'}\right)^{N'-1} \geq \frac{1}{eN'}$$

for $N' > 1$. For $N' = 1$, they hold with probability 1.

In this case, only u^* uses pk^*, and all other users' keys and signatures are generated using f_1 and f_2. Since the challenge key and the signatures returned by SIGN′ have the same distributions as in the

Reduction $\mathcal{B}^{\text{Sign}',\text{Forge}'}(\mathsf{par}, \mathsf{pk}^*)$

00 $f_1 \leftarrow \mathcal{F}(\mathcal{U}, \mathcal{R}_{\mathtt{k}})$
01 $f_2 \leftarrow \mathcal{F}(\mathcal{U} \times \mathcal{M}, \mathcal{R}_{\mathtt{s}})$
02 $f_3 \leftarrow \mathcal{F}(\mathcal{U}, [N'])$
03 $\mathsf{win} = \textbf{false}$
04 $\mathcal{S}_{\mathtt{cor}} := \emptyset$
05 run $\mathcal{A}^{\text{Pk},\text{Cor},\text{Sign},\text{Forge}}(\mathsf{par})$
06 **abort**

Oracle $\text{Pk}(u)$

07 $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{GetKey}(u)$
08 **return** pk

Oracle $\text{Cor}(u)$

09 $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{GetKey}(u)$
10 **if** $f_3(u) = 1$: **abort**
11 **return** $(\mathsf{pk}, \mathsf{sk})$

Alg $\mathsf{GetKey}(u)$

12 **if** $f_3(u) = 1$: **return** $(\mathsf{pk}^*, \bot)$
13 $(\mathsf{pk}_u, \mathsf{sk}_u) := \mathsf{KG}(; f_1(u))$
14 **return** $(\mathsf{pk}_u, \mathsf{sk}_u)$

Oracle $\text{Sign}(u, m)$

15 $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{GetKey}(u)$
16 **if** $f_3(u) = 1$
17 **return** $\sigma \leftarrow \text{Sign}'(m)$
18 **return** $\mathsf{Sign}(\mathsf{sk}, m; f_2(u, m))$

Oracle $\text{Forge}(u, m, \sigma)$

19 $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{GetKey}(u)$
20 **if** $f_3(u) = 1 \wedge \mathsf{Vrfy}(\mathsf{pk}, m, \sigma) = 1$
21 $\text{Forge}'(m, \sigma)$

Fig. 12. Reduction $\mathcal{B}$ for Lemma 10.

original game, the probability that $\mathcal{B}$ produces each such sequence with these f_3 values is at least $1/(eN')$ times its probability in the original game. This also covers users chosen by $\mathcal{A}$ based on earlier answers. Moreover, u^* has not been corrupted, so $\mathcal{B}$ does not abort before this forgery. Only u^*'s signing queries are forwarded to Sign', so (m^*, σ^*) is also fresh in the single-user game. Thus, the forwarded Forge' query sets its winning flag to **true**, and subsequent queries do not reset this flag. Summing over the successful executions of the original game, we have

$$\Pr\left[\mathsf{sEUF\text{-}1CMA}^{\mathcal{B}}_{\mathsf{SIG}} \Rightarrow \textbf{true}\right] \geq \frac{1}{eN'} \Pr\left[\mathsf{MU}^c\text{-}\mathsf{sEUF\text{-}1CMA}^{\mathcal{A}}_{\mathsf{SIG}} \Rightarrow \textbf{true}\right]$$
$$\implies \mathbf{Adv}^{\mathsf{MU}^c\text{-}\mathsf{sEUF\text{-}1CMA}}_{\mathsf{SIG}}(\mathcal{A}) \leq eN' \cdot \mathbf{Adv}^{\mathsf{sEUF\text{-}1CMA}}_{\mathsf{SIG}}(\mathcal{B}).$$

and the running time and memory usage of $\mathcal{B}$ are

$$\mathbf{T}^*(\mathcal{B}) = \mathbf{T}(\mathcal{A}) + \mathrm{O}(Q_{\mathrm{Pk}} + Q_{\mathtt{forge}} + Q_{\mathtt{sign}} + Q_{\mathtt{cor}}) \cdot \mathbf{T}(\mathsf{SIG})$$
$$\mathbf{Mem}^*(\mathcal{B}) = \mathbf{Mem}(\mathcal{A}) + \mathbf{Mem}(\mathsf{SIG}) + \log|\mathcal{U}| + \log N'$$

When applying this lemma to the AKE reduction, choose N' as an a priori bound on all users that reduction can touch, including users introduced through hash and message-checking queries. Repeated key-generation calls contribute to running time even though they do not increase the number of distinct users.

Game $\mathsf{IND\text{-}CPA}_{\mathsf{PKE},b}$	**Oracle** $\textsc{ChalPk}(u)$	**Oracle** $\textsc{ChalCt}(u,i,m)$
00 $\mathsf{par} \leftarrow \mathsf{Setup}$	03 **if** $\mathbf{pk}[u] = \bot$:	06 **if** $\mathbf{c}[u,i] \neq \bot$: **return** $\mathbf{c}[u,i]$
01 $b' \leftarrow \mathcal{A}^{\textsc{ChalPk},\textsc{ChalCt}}(\mathsf{par})$	04 $(\mathbf{pk}[u], \mathbf{sk}[u]) \leftarrow \mathsf{KG}$	07 $m_0 := m, m_1 := 0^{\lvert m \rvert}$
02 **return** b'	05 **return** $\mathbf{pk}[u]$	08 $c \leftarrow \mathsf{Enc}(\textsc{ChalPk}(u), m_b)$
		09 $\mathbf{c}[u,i] := c$
		10 **return** $\mathbf{c}[u,i]$

Fig. 13. Security game for defining IND-CPA security. We implicitly assume that all user identifiers belong to $\mathcal{U}$.

E Generic Construction from IND-CPA PKE to OW-PCA KEM

Before showing our generic construction, we first recall the syntax of public key encryption and its security definition.

We first define two notions of correctness for PKE schemes. The first one is a weak notion, which only requires decryption correctness for a uniformly random message; this is sufficient for our KEM construction, since encapsulation itself samples the underlying message at random. The second one is a stronger notion, which requires decryption correctness for every message, even if it is chosen by an unbounded adversary. Note that the latter is mainly used in security proofs, for example to bound the probability of attacks that exploit decryption errors, and is not needed for the correctness of the KEM itself.

Definition 9 (PKE Correctness under Random Messages). *Let* $\mathsf{PKE} := (\mathsf{Setup}, \mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ *be a PKE scheme with message space* $\mathcal{M}$*. We say* PKE *is* $(1 - \delta_{\mathsf{rm}})$*-correct under random messages if*

$$\Pr\left[\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(\mathsf{pk}, m)) = m : m \leftarrow \mathcal{M}\right] \geq 1 - \delta_{\mathsf{rm}},$$

where the probability is taken over $\mathsf{par} \leftarrow \mathsf{Setup}$*,* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}$*, the random choice of* m*, and the randomness of* Enc*. Equivalently, we say that the decryption error probability under random messages is at most* δ_{rm}*.*

Definition 10 ((Strong) PKE Correctness [HHK17]). *Let* $\mathsf{PKE} := (\mathsf{Setup}, \mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ *be a PKE scheme. We say* PKE *is* $(1 - \delta)$*-correct if, for all adversaries (that can be unbounded)* $\mathcal{A}$*, we have*

$$\Pr\left[\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(\mathsf{pk}, m)) = m : m \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{sk})\right] \geq 1 - \delta,$$

where the probability is taken over $\mathsf{par} \leftarrow \mathsf{Setup}, (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}$*, and the randomness of* Enc*. When using this definition in a security proof, we may use* δ *to bound the probability of decryption error.*

It is straightforward to see that $(1 - \delta)$-correctness implies $(1 - \delta)$-correctness under random messages, but the converse is not necessarily true.

Definition 11 (γ-Spreadness of PKE [HHM22]). *Let* $\mathsf{PKE} := (\mathsf{Setup}, \mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ *be a PKE scheme. We say* PKE *is* γ*-spread if, for any* $\mathsf{par} \leftarrow \mathsf{Setup}, (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}$*, and* $m \in \mathsf{PKE}.\mathcal{M}$*, we have*

$$\max_{\mathsf{c} \in \mathsf{PKE}.\mathcal{C}} \Pr\left[\mathsf{Enc}(\mathsf{pk}, m) = \mathsf{c}\right] \leq 2^{-\gamma},$$

where the probability is taken over the randomness of Enc*.*

Definition 12 (IND-CPA Security of PKE). *Let* $\mathsf{PKE} := (\mathsf{Setup}, \mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ *be a PKE scheme. Consider the game* IND-CPA *defined in Figure 13. We say* PKE *is IND-CPA secure if, for all PPT adversaries* $\mathcal{A}$*, the following advantage is negligible on the security parameter:*

$$\mathbf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{PKE}}(\mathcal{A}) := \left|\Pr\left[\mathsf{IND\text{-}CPA}^{\mathcal{A}}_{\mathsf{PKE},0} \Rightarrow 1\right] - \Pr\left[\mathsf{IND\text{-}CPA}^{\mathcal{A}}_{\mathsf{PKE},1} \Rightarrow 1\right]\right|$$

Construction. Let $\mathsf{PKE} := (\mathsf{Setup}, \mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ be a PKE scheme with message space $\mathcal{M}$, randomness space $\mathcal{R}$, and ciphertext space $\mathcal{C}$. Let ℓ and l be some integers. Let $\mathsf{G} : \{0,1\}^{\ell} \times \mathcal{M} \to \mathcal{R}$, $\mathsf{H} : \{0,1\}^{\ell} \times \mathcal{M} \times \mathcal{C} \to \mathcal{K}$, and $\mathsf{H}' : \{0,1\}^{\ell} \times \{0,1\}^{l} \times \mathcal{C} \to \mathcal{K}$ be three hash functions (that will be modeled as random oracles in the security proof). We construct a generic construction in Figure 14.

Alg KG	**Alg** Encaps$((\mathsf{pk}, \alpha))$	**Alg** Decaps$((\mathsf{pk},\mathsf{sk},\mathsf{s},\alpha), c)$
00 $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{PKE.KG}$	06 $m \leftarrow \mathcal{M}$	11 $(\mathsf{pk},\mathsf{sk},\mathsf{s},\alpha) := \mathsf{sk}_{\mathtt{kem}}$
01 $\alpha \leftarrow \{0,1\}^{\ell}$	07 $r := \mathsf{G}(\alpha, m)$	12 $m' := \mathsf{Dec}(\mathsf{sk}, c)$
02 $\mathsf{s} \leftarrow \{0,1\}^{l}$	08 $c := \mathsf{Enc}(\mathsf{pk}, m; r)$	13 **if** $m' = \bot \ \vee \ c \neq \mathsf{Enc}(\mathsf{pk}, m'; \mathsf{G}(\alpha, m'))$
03 $\mathsf{pk}_{\mathtt{kem}} := (\mathsf{pk}, \alpha)$	09 $k := \mathsf{H}(\alpha, m, c)$	14 $\quad k := \mathsf{H}'(\alpha, \mathsf{s}, c)$
04 $\mathsf{sk}_{\mathtt{kem}} := (\mathsf{pk},\mathsf{sk},\mathsf{s},\alpha)$	10 **return** (c, k)	15 **else** $k := \mathsf{H}(\alpha, m', c)$
05 **return** $(\mathsf{pk}_{\mathtt{kem}}, \mathsf{sk}_{\mathtt{kem}})$		16 **return** k

Fig. 14. A generic KEM construction KEM based on a PKE scheme PKE. The setup algorithm KEM.Setup is the same as the setup algorithm of PKE except that the parameter of KEM also includes ℓ and l. The secret key includes the PKE public key needed for the re-encryption check; this adds one public key to its representation.

Theorem 3. *Let* KEM *be the KEM construction presented in Figure 14. Assume* $|\mathcal{M}| \geq |\mathcal{U}| \cdot |\mathcal{I}|$ *and* $2^{\ell} \geq |\mathcal{U}|$*, so that the injections used below exist. If the PKE scheme* PKE *is* $(1-\delta)$*-correct and* γ*-spread, then for any adversary* $\mathcal{A}$ *against the OW-PCA security of* KEM*, there exists an adversary* $\mathcal{B}$ *such that*

$$\begin{aligned}
&\mathbf{Adv}^{\mathsf{OW\text{-}PCA}}_{\mathsf{KEM}}(\mathcal{A}) \\
&\leq \mathbf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{PKE}}(\mathcal{B}) + (Q_{\mathrm{PK}} + N')(N' + Q_{\mathsf{G}}) \cdot \delta + \frac{Q_{\mathsf{H}} + (Q_{\mathsf{G}} + N')^2}{2^{\gamma}} \\
&\quad + \frac{3(Q_{\mathrm{PCO}} + 1)}{|\mathcal{K}|} + \frac{Q_{\mathtt{H}} \cdot |\mathcal{U}| \cdot |\mathcal{I}| + Q_{\mathsf{G}} \cdot |\mathcal{I}| + (Q_{\mathrm{CT}} + 1)^2}{|\mathcal{M}|} \\
&\quad + \frac{2(Q_{\mathrm{PCO}} + Q_{\mathsf{H}'} + 1)}{2^{l}} + \frac{(N' + Q_{\mathrm{PK}})^2}{2^{\ell+1}}
\end{aligned}$$

where $\mathbf{T}(\mathsf{PKE})$ *and* $\mathbf{Mem}(\mathsf{PKE})$ *are the running time and memory usage of one execution of* PKE*, respectively, and*

- $\mathcal{A}$ *issues at most* Q_{PK}, Q_{CT}, Q_{PCO},Q_{G}, Q_{H}, *and* $Q_{\mathsf{H}'}$ *direct queries to oracles* ChalPk, ChalCt, Pco,G, H, *and* H′*, respectively, and* $N' = Q_{\mathsf{H}} + Q_{\mathrm{CT}} + Q_{\mathrm{PCO}} + 1$*;*
- $\mathcal{B}$ *issues at most* $Q_{\mathrm{PK}} + N'$ *and* $Q_{\mathrm{CT}} + 1$ *queries to the* ChalPk *and* ChalCt *oracles of the* IND-CPA *game, respectively.*
- $\mathbf{T}^*(\mathcal{B}) = \mathbf{T}^*(\mathcal{A}) + \mathrm{O}(Q) + \mathrm{O}(Q_{\mathsf{H}}) \cdot \mathbf{T}(\mathsf{PKE})$
- $\mathbf{Mem}^*(\mathcal{B}) = \mathbf{Mem}^*(\mathcal{A}) + \log|\mathcal{U}| + \log|\mathcal{I}| + \mathrm{O}(\mathbf{Mem}(\mathsf{PKE}))$

Moreover, if PKE *is* $(1-\delta_{\mathsf{rm}})$*-correct under random messages (cf. Definition 9), then* KEM *is* $(1-\delta_{\mathsf{rm}})$*-correct.*

We combine techniques from IND-CCA proof in the QROM [JZC+18,SXY18] and memory-tight PKE constructions in [JK22] to prove Theorem 3. Before presenting the proof, we first give some remarks on the comparison with prior work.

Remark 6 (Comparison with the result in [JK22, Section 5.2]). In [JK22], Jaeger et al. show that, by combining their memory-tight T[IND-CPA → OW-PCA] and aV[OW-PCA → OW-PCVA] transformations, one can get a OW-PCVA KEM scheme from an IND-CPA PKE scheme in the multi-user, multi-session setting. Such an OW-PCVA KEM can be also used to instantiate our AKE construction. The main difference between their result and ours lies in the mechanism of rejecting invalid ciphertexts. Jaeger et al.'s T[IND-CPA → OW-PCA] construction considers *explicit rejection*, so only having the Pco oracle is not sufficient to simulate the AKE game since we still need a ciphertext-validity oracle (CVO) to explicitly exclude invalid ciphertexts. To add such a CVO, we need to apply aV to the OW-PCA PKE to obtain an OW-PCVA KEM. However, the aV transformation introduces an extra key confirmation tag, which expands the length of the KEM ciphertext. This is the main motivation for us to propose a new OW-PCA KEM construction with implicit rejection, which can get rid of the ciphertext-validity oracle and thus preserve the length of the underlying PKE ciphertext.

Moreover, the correctness bound of Jaeger et al.'s OW-PCVA constructions can be improved (i.e., remove the multiplicative factor Q_{G}) by simply using the weaker correctness definition of PKE in Definition 9 instead of the stronger one in Definition 10, since the correctness of the resulting OW-PCVA KEM only depends on the correctness of the intermediate OW-PCA scheme, which can be directly bounded by the correctness of the underlying CPA-secure PKE under random messages.

Game $\mathbf{G}_0$-$\mathbf{G}_7$

00 $f_{\mathtt{kg}} \leftarrow \mathsf{Func}(\mathcal{U}, \mathcal{R}_{\mathtt{ekg}})$ // $\mathbf{G}_1$-$\mathbf{G}_7$
01 $f_{\mathtt{prfk}} \leftarrow \mathsf{Func}(\mathcal{U}, \{0,1\}^l)$ // $\mathbf{G}_1$-$\mathbf{G}_7$
02 $f_{\mathtt{G}} \leftarrow \mathsf{Func}(\{0,1\}^\ell \times \mathcal{M}, \mathcal{R})$ // $\mathbf{G}_1$-$\mathbf{G}_7$
03 $f_{\mathtt{H}} \leftarrow \mathsf{Func}(\{0,1\}^\ell \times \mathcal{M} \times \mathcal{C}, \mathcal{K})$ // $\mathbf{G}_1$-$\mathbf{G}_7$
04 $f_{\mathtt{h}} \leftarrow \mathsf{Func}(\{0,1\}^\ell \times \{0,1\}^l \times \mathcal{C}, \mathcal{K})$ // $\mathbf{G}_1$-$\mathbf{G}_7$
05 $(h_{\mathtt{msg}}, h_{\mathtt{msg}}^{-1}) \leftarrow \mathsf{InjF}(\mathcal{U} \times \mathcal{I}, \mathcal{M})$ // $\mathbf{G}_1$-$\mathbf{G}_7$
06 $(h_{\mathtt{tag}}, h_{\mathtt{tag}}^{-1}) \leftarrow \mathsf{InjF}(\mathcal{U}, \{0,1\}^\ell)$ // $\mathbf{G}_1$-$\mathbf{G}_7$
07 $f_{\mathtt{Hkey}} \leftarrow \mathsf{Func}(\mathcal{U} \times \mathcal{C}, \mathcal{K})$ // $\mathbf{G}_2$-$\mathbf{G}_7$
08 $f_{\mathtt{msg}} \leftarrow \mathsf{Func}(\mathcal{U} \times \mathcal{I}, \mathcal{R})$ // $\mathbf{G}_5$-$\mathbf{G}_6$
09 $\mathsf{par} \leftarrow \mathsf{Setup}$
10 $O := (\textsc{ChalPk}, \textsc{ChalCt}, \textsc{Pco})$
11 $(u^*, i^*, k^*) \leftarrow \mathcal{A}^O(\mathsf{par})$
12 **return** $\textsc{Pco}(u^*, \textsc{ChalCt}(u^*, i^*), k^*)$

Oracle $\textsc{ChalPk}(u)$

13 **if** $\mathbf{pk}[u] \neq \bot$ // $\mathbf{G}_0$
14 $(\mathsf{pk}_u, \mathsf{sk}_u, \mathsf{s}_u, \alpha_u) := \mathbf{sk}[u]$ // $\mathbf{G}_0$
15 **else** // $\mathbf{G}_0$
16 $(\mathsf{pk}_u, \mathsf{sk}_u) \leftarrow \mathsf{PKE.KG}$ // $\mathbf{G}_0$
17 $\alpha_u \leftarrow \{0,1\}^\ell$, $\mathsf{s}_u \leftarrow \{0,1\}^l$ // $\mathbf{G}_0$
18 $\mathbf{pk}[u] := (\mathsf{pk}_u, \alpha_u)$ // $\mathbf{G}_0$
19 $\mathbf{sk}[u] := (\mathsf{pk}_u, \mathsf{sk}_u, \mathsf{s}_u, \alpha_u)$ // $\mathbf{G}_0$
20 $(\mathsf{pk}_u, \mathsf{sk}_u) := \mathsf{PKE.KG}(; f_{\mathtt{kg}}(u))$ // $\mathbf{G}_1$-$\mathbf{G}_7$
21 $\alpha_u := h_{\mathtt{tag}}(u)$, $\mathsf{s}_u := f_{\mathtt{prfk}}(u)$ // $\mathbf{G}_1$-$\mathbf{G}_7$
22 **return** $(\mathsf{pk}_u, \alpha_u)$

Oracle $\textsc{ChalCt}(u, i)$

23 $\textsc{ChalPk}(u)$
24 **if** $\mathbf{c}[u,i] \neq \bot$: **return** $\mathbf{c}[u,i]$ // $\mathbf{G}_0$
25 $m \leftarrow \mathcal{M}$ // $\mathbf{G}_0$
26 $m := h_{\mathtt{msg}}(u,i)$ // $\mathbf{G}_1$-$\mathbf{G}_7$
27 $c := \mathsf{Enc}(\mathsf{pk}_u, m; \mathsf{G}(\alpha_u, m))$ // $\mathbf{G}_0$-$\mathbf{G}_4$
28 $c := \mathsf{Enc}(\mathsf{pk}_u, m; f_{\mathtt{msg}}(u,i))$ // $\mathbf{G}_5$-$\mathbf{G}_6$
29 $c := \mathsf{Enc}(\mathsf{pk}_u, 0^{|m|}; f_{\mathtt{msg}}(u,i))$ // $\mathbf{G}_7$
30 $k := \mathsf{H}(\alpha_u, m, c)$ // $\mathbf{G}_0$-$\mathbf{G}_2$
31 $k := f_{\mathtt{Hkey}}(u,c)$ // $\mathbf{G}_3$-$\mathbf{G}_7$
32 $\mathbf{c}[u,i] := c$ // $\mathbf{G}_0$
33 **return** c

Oracle $\textsc{Pco}(u, c, k)$

34 $\textsc{ChalPk}(u)$
35 $m' := \mathsf{Dec}(\mathsf{sk}_u, c)$ // $\mathbf{G}_0$-$\mathbf{G}_3$
36 **if** $m' = \bot \vee c \neq \mathsf{Enc}(\mathsf{pk}_u, m'; \mathsf{G}(\alpha_u, m'))$ // $\mathbf{G}_0$-$\mathbf{G}_3$
37 $k' := \mathsf{H}'(\alpha_u, \mathsf{s}_u, c)$ // $\mathbf{G}_0$-$\mathbf{G}_3$
38 **else** $k' := \mathsf{H}(\alpha_u, m', c)$ // $\mathbf{G}_0$-$\mathbf{G}_3$
39 **if** $k = k'$: **return** 1 // $\mathbf{G}_0$-$\mathbf{G}_3$
40 $k'_0 = f_{\mathtt{Hkey}}(u,c)$ // $\mathbf{G}_4$-$\mathbf{G}_7$
41 $k'_1 = \mathsf{H}'(\alpha_u, \mathsf{s}_u, c)$ // $\mathbf{G}_4$-$\mathbf{G}_7$
42 **if** $k = k'_0 \ \vee \ k = k'_1$: **return** 1 // $\mathbf{G}_4$-$\mathbf{G}_7$
43 **return** 0

Oracle $\mathsf{G}(\alpha, m)$

44 **if** $\mathcal{L}_{\mathsf{G}}[\alpha, m] = \bot$ // $\mathbf{G}_0$
45 $\mathcal{L}_{\mathsf{G}}[\alpha, m] := r \leftarrow \mathcal{R}$ // $\mathbf{G}_0$
46 **return** $\mathcal{L}_{\mathsf{G}}[\alpha, m]$ // $\mathbf{G}_0$
47 $(u, i) := h_{\mathtt{msg}}^{-1}(m)$ // $\mathbf{G}_5$-$\mathbf{G}_7$
48 **if** $(u,i) \neq \bot \ \wedge \ \alpha = h_{\mathtt{tag}}(u)$ // $\mathbf{G}_5$-$\mathbf{G}_7$
49 **return** $f_{\mathtt{msg}}(u,i)$ // $\mathbf{G}_5$
50 **abort** and **output** 1 // $\mathbf{G}_6$-$\mathbf{G}_7$
51 **return** $f_{\mathtt{G}}(\alpha, m)$ // $\mathbf{G}_1$-$\mathbf{G}_7$

Oracle $\mathsf{H}(\alpha, m, c)$

52 **if** $\mathcal{L}_{\mathsf{H}}[\alpha, m, c] = \bot$ // $\mathbf{G}_0$
53 $\mathcal{L}_{\mathsf{H}}[\alpha, m, c] := k \leftarrow \mathcal{K}$ // $\mathbf{G}_0$
54 **return** $\mathcal{L}_{\mathsf{H}}[\alpha, m, c]$ // $\mathbf{G}_0$
55 **if** $h_{\mathtt{msg}}^{-1}(m) \neq \bot$ // $\mathbf{G}_6$-$\mathbf{G}_7$
56 **abort** and **output** 1 // $\mathbf{G}_6$-$\mathbf{G}_7$
57 **if** $\bot \neq u := h_{\mathtt{tag}}^{-1}(\alpha)$ // $\mathbf{G}_2$-$\mathbf{G}_7$
58 $\textsc{ChalPk}(u)$ // $\mathbf{G}_2$-$\mathbf{G}_7$
59 **if** $m \neq \bot$ // $\mathbf{G}_2$-$\mathbf{G}_7$
60 $\wedge \ c = \mathsf{Enc}(\mathsf{pk}_u, m; \mathsf{G}(\alpha, m))$ // $\mathbf{G}_2$-$\mathbf{G}_7$
61 **return** $f_{\mathtt{Hkey}}(u,c)$ // $\mathbf{G}_2$-$\mathbf{G}_7$
62 **return** $f_{\mathtt{H}}(\alpha, m, c)$ // $\mathbf{G}_1$-$\mathbf{G}_7$

Oracle $\mathsf{H}'(\alpha, \mathsf{s}, c)$

63 **if** $\mathcal{L}_{\mathsf{H}'}[\alpha, \mathsf{s}, c] = \bot$ // $\mathbf{G}_0$
64 $\mathcal{L}_{\mathsf{H}'}[\alpha, \mathsf{s}, c] := k \leftarrow \mathcal{K}$ // $\mathbf{G}_0$
65 **return** $\mathcal{L}_{\mathsf{H}'}[\alpha, \mathsf{s}, c]$ // $\mathbf{G}_0$
66 **return** $f_{\mathtt{h}}(\alpha, \mathsf{s}, c)$ // $\mathbf{G}_1$-$\mathbf{G}_7$

Fig. 15. Security game for defining OW-PCA security. We implicitly assume that all user identifiers belong to $\mathcal{U}$. Moreover, for simplicity, we assume that pk_u is from $(\mathsf{pk}_u, \mathsf{sk}_u) := \mathsf{PKE.KG}(; f_{\mathtt{kg}}(u))$, $\alpha_u := h_{\mathtt{tag}}(u)$, and $\mathsf{s}_u := f_{\mathtt{prfk}}(u)$ since $\mathbf{G}_1$. This does not affect the memory-tightness since we can recompute them on-the-fly.

Proof (Theorem 3). For correctness, Encaps samples m uniformly from $\mathcal{M}$, and $\mathsf{G}(\alpha, m)$ gives uniform encryption randomness in the random oracle model. If PKE decryption recovers m, then the re-encryption check passes and Decaps returns the original KEM key. Hence, the KEM decryption error probability is at most δ_{rm}.

We use the game sequence in Figure 15 to prove Theorem 3, where $\mathbf{G}_0$ is the original OW-PCA game.

$$\Pr\left[\mathsf{OW\text{-}PCA}^{\mathcal{A}}_{\mathsf{KEM}} \Rightarrow 1\right] = \Pr\left[\mathbf{G}_0^{\mathcal{A}} \Rightarrow 1\right]$$

Game $\mathbf{G}_1$ (Use RFs/RPs to derive randomness). In this game, we use random functions/injections to derive all randomness used in $\textsc{ChalPk}$, $\textsc{ChalCt}$, G, H, and H'. Specifically,

- Random oracles G, H, and H' are simulated (without recording) by functions $f_{\mathtt{G}}, f_{\mathtt{H}}, f_{\mathtt{h}}$, respectively.
- We use random functions $f_{\mathtt{prfk}}$ and $f_{\mathtt{kg}}$ to simulate the implicit-rejection secret key s and the key pair $(\mathsf{pk}, \mathsf{sk})$ in oracle $\textsc{ChalPk}$.
- We use random injection $h_{\mathtt{msg}}$ to generate the messages in $\textsc{ChalCt}$ (cf. line 26) and use random injection $h_{\mathtt{tag}}$ to generate random strings of challenge public keys (cf. line 21).

The first change is conceptual. For the generation of the random strings in ChalPk and the challenge messages in ChalCt, since we use a random injective function (i.e. sampling without replacement) instead of uniformly random sampling (i.e., sampling with replacement), by standard birthday bounds (one for ChalPk and another for ChalCt), the probability difference introduced by this change is at most

$$\left|\Pr\left[\mathbf{G}_0^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_1^{\mathcal{A}} \Rightarrow 1\right]\right| \leq \frac{(N' + Q_{\mathrm{PK}})^2}{2^{\ell+1}} + \frac{(Q_{\mathrm{CT}} + 1)^2}{2|\mathcal{M}|},$$

where $N' \leq Q_{\mathrm{PCO}} + Q_{\mathrm{CT}} + Q_{\mathsf{H}} + 1$ is the maximal number of distinct public keys generated during the game excluding the queries to ChalPk (i.e., indirect queries to ChalPk).

Game $\mathbf{G}_2$ and $\mathbf{G}_3$ (Compute KEM keys without messages). In game $\mathbf{G}_2$, we let $\mathsf{H}(\alpha, m, \mathsf{c}) := f_{\mathtt{Hkey}}(u, \mathsf{c})$ if $u = h_{\mathtt{tag}}^{-1}(\alpha)$ and $\mathsf{c} = \mathsf{Enc}(\mathsf{pk}_u, m; \mathsf{G}(\alpha, m))$, where $f_{\mathtt{Hkey}}$ is a random function (cf. lines 57 to 61). Then, in game $\mathbf{G}_3$, we compute the challenge KEM key k in ChalCt as $k := f_{\mathtt{Hkey}}(u, \mathsf{c})$.

We first bound the probability difference caused by the changes introduced in $\mathbf{G}_2$. In $\mathbf{G}_2$, we program those queries $\mathsf{H}(\alpha, m, \mathsf{c})$ only if α is the random string of some challenge public key (i.e., $\exists u \in \mathcal{U}$ such that $\alpha = h_{\mathtt{tag}}(u)$), thus we can rewrite the random oracle H in the following form:

$$\mathsf{H}(\alpha, m, \mathsf{c}) = \begin{cases} f_{\mathtt{Hkey}}(u, \mathsf{c}), \text{where } u = h_{\mathtt{tag}}^{-1}(\alpha) & \text{if } \mathsf{c} = \mathsf{Enc}(\mathsf{pk}_u, m; \mathsf{G}(\alpha, m)), \\ f_{\mathtt{H}}(\alpha, m, \mathsf{c}), & \text{otherwise.} \end{cases}$$

Since $h_{\mathtt{tag}}$ is an injection, the only way to find a collision of H is by finding some (α, m, c) and (α, m', c) such that $m \neq m'$ and $\mathsf{Enc}(\mathsf{pk}_u, m; \mathsf{G}(\alpha, m)) = \mathsf{c} = \mathsf{Enc}(\mathsf{pk}_u, m'; \mathsf{G}(\alpha, m'))$. Therefore, to prove that H in $\mathbf{G}_2$ still behaves as a random oracle, it suffices to show that the function $\mathsf{Enc}(\mathsf{pk}_u, \cdot; \mathsf{G}(\alpha, \cdot))$ behaves as an injective function in $\mathcal{A}$'s view for any pk_u (that is generated by ChalPk and bound to α via $h_{\mathtt{tag}}$).

We use the γ-spreadness definition in Definition 11 to show that, if G is a classical[1] random oracle, then $\mathsf{Enc}(\mathsf{pk}, \cdot; \mathsf{G}(\alpha, \cdot))$ is injective in $\mathcal{A}$'s view. Suppose that $\mathcal{A}$ queried H on some m and m' where $\mathsf{Enc}(\mathsf{pk}, m; \mathsf{G}(\alpha, m)) = \mathsf{Enc}(\mathsf{pk}, m'; \mathsf{G}(\alpha, m'))$. Let $m \neq m'$ be two arbitrary messages and pk be an arbitrary honestly generated public key of PKE. We fix $c := \mathsf{Enc}(\mathsf{pk}, m; \mathsf{G}(\alpha, m))$ as a target and consider the probability that $\mathsf{Enc}(\mathsf{pk}, m'; \mathsf{G}(\alpha, m')) = c$. Since G is a random oracle, a fresh query to G will produce a uniformly random value in $\mathcal{R}$ (note that all our arguments here are in the classical ROM and the access to G is in a black-box way). By the γ-spreadness of PKE, the probability that $\mathsf{Enc}(\mathsf{pk}, m'; \mathsf{G}(\alpha, m')) = c$ is bounded by $2^{-\gamma}$. Moreover, since there are at most $Q_{\mathsf{G}} + Q_{\mathsf{H}} + Q_{\mathrm{CT}} + Q_{\mathrm{PCO}} + 1 (= Q_{\mathsf{G}} + N')$ queries to G during the game (including all indirect queries), there are at most $(Q_{\mathsf{G}} + N')$ messages that can be queried to G. By a union bound (over all message pairs), we have

$$\left|\Pr\left[\mathbf{G}_1^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_2^{\mathcal{A}} \Rightarrow 1\right]\right| \leq (Q_{\mathsf{G}} + N')^2 \cdot 2^{-\gamma-1}.$$

In game $\mathbf{G}_2$, we have $k := \mathsf{H}(\alpha_u, m, \mathsf{c}) = f_{\mathtt{Hkey}}(u, \mathsf{c})$ because $\mathsf{H}(\alpha, m, \mathsf{c})$ is set as $f_{\mathtt{Hkey}}(u, \mathsf{c})$ if $u = h_{\mathtt{tag}}^{-1}(\alpha)$ and $\mathsf{c} = \mathsf{Enc}(\mathsf{pk}_u, m; \mathsf{G}(\alpha, m))$. Hence, the change introduced in $\mathbf{G}_3$ is conceptual:

$$\Pr\left[\mathbf{G}_2^{\mathcal{A}} \Rightarrow 1\right] = \Pr\left[\mathbf{G}_3^{\mathcal{A}} \Rightarrow 1\right].$$

Game $\mathbf{G}_4$ (Simulate PCO without sk). In this game, we no longer use secret keys to simulate the Pco oracle. Instead, to determine whether a KEM key k is the decapsulated key of a ciphertext c with respect to pk_u, we only check if $k = f_{\mathtt{Hkey}}(u, \mathsf{c})$ or $k = \mathsf{H}'(\alpha_u, \mathsf{s}_u, \mathsf{c})$ (cf. lines 40 to 41). The invalid message check $m \neq \perp$ and re-encryption check $c = \mathsf{Enc}(\mathsf{pk}_u, m; \mathsf{G}(\alpha_u, m))$ are removed.

We first analyze the output distributions of Pco in $\mathbf{G}_3$ and $\mathbf{G}_4$. In $\mathbf{G}_4$, Pco always returns $[\![k = \mathsf{H}'(\alpha_u, \mathsf{s}_u, c) \vee k = f_{\mathtt{Hkey}}(u, \mathsf{c})]\!]$. Suppose that $\mathcal{A}$ queries $\mathrm{Pco}(u, \mathsf{c}, k)$ and let $m' := \mathsf{Dec}(\mathsf{sk}_u, c)$ be the decapsulated message in $\mathbf{G}_3$. We consider the following two cases to analyze the output distributions of Pco in $\mathbf{G}_3$ and $\mathbf{G}_4$.

- *Case (1): $m' = \perp$ or $\mathsf{c} \neq \mathsf{Enc}(\mathsf{pk}_u, m', \mathsf{G}(\alpha_u, m'))$.* In $\mathbf{G}_3$, Pco returns $[\![k = \mathsf{H}'(\alpha_u, \mathsf{s}_u, c)]\!]$ in this case. In $\mathbf{G}_4$, Pco returns $[\![k = \mathsf{H}'(\alpha_u, \mathsf{s}_u, c) \vee k = f_{\mathtt{Hkey}}(u, \mathsf{c})]\!]$. To distinguish $\mathbf{G}_3$ and $\mathbf{G}_4$ in this case, $\mathcal{A}$

[1] We emphasize *classical random oracle* because this does not apply to a proof in the quantum random oracle model.

needs to query $\textsc{Pco}(u, \mathsf{c}, f_{\mathtt{Hkey}}(u, \mathsf{c}))$. However, the only way to get information about $f_{\mathtt{Hkey}}(u, \mathsf{c})$ is to query $\mathsf{H}(\alpha_u, m, \mathsf{c})$ on some $m \neq \bot$ such that $\mathsf{c} = \mathsf{Enc}(\mathsf{pk}_u, m; \mathsf{G}(\alpha_u, m))$.

Here we need to use the correctness of PKE (cf. Definition 10) to analyze the probability that $k = f_{\mathtt{Hkey}}(u, \mathsf{c})$ in this case. Note that s_u is generated uniformly at random in both games; the probability that $k = \mathsf{H}'(\alpha_u, \mathsf{s}_u, c)$ is the same in both games. By the change made in $\mathbf{G}_2$, we have that $\mathcal{A}$ knows $f_{\mathtt{Hkey}}(u, \mathsf{c})$ only if $\mathcal{A}$ queried $\mathsf{H}(\alpha_u, m, \mathsf{c})$ on some m such that $\mathsf{c} = \mathsf{Enc}(\mathsf{pk}_u, m; \mathsf{G}(\alpha_u, m)) \neq \mathsf{Enc}(\mathsf{pk}_u, m', \mathsf{G}(\alpha_u, m'))$, where $m' = \mathsf{Dec}(\mathsf{sk}_u, c)$. Hence, this case happens only if $\mathcal{A}$ finds some m that causes decryption error.

There are at most $Q_{\mathrm{PK}} + N'$ distinct public keys, including those exposed by direct $\textsc{ChalPk}$ queries, and at most $N' + Q_{\mathsf{G}}$ fresh inputs to G, including indirect queries. We guess the target key among all first key generations and the first query of the relevant G input. Applying the strong correctness definition of PKE gives an error bound of $(Q_{\mathrm{PK}} + N')(N' + Q_{\mathsf{G}})\delta$. This is only a statistical argument: the correctness adversary is unrestricted and may keep tables, so no memory-tight guessing reduction is needed.

Without a corresponding H query, $\mathcal{A}$ may still guess $f_{\mathtt{Hkey}}(u, \mathsf{c})$. Counting the first successful guess over all $\textsc{Pco}$ queries and the final output gives a bound of $(Q_{\mathrm{PCO}} + 1)/|\mathcal{K}|$.

– *Case (2): $m' \neq \bot$ and $\mathsf{c} = \mathsf{Enc}(\mathsf{pk}_u, m', \mathsf{G}(\alpha_u, m'))$.* The answers differ if $\mathcal{A}$ supplies the rejection key $\mathsf{H}'(\alpha_u, \mathsf{s}_u, c)$ rather than the normal key. The probability of finding the secret seed through H' queries is bounded by $(Q_{\mathrm{PCO}} + Q_{\mathsf{H}'} + 1)/2^l$. Without querying this oracle input, $\mathcal{A}$ can still guess its output key; all $\textsc{Pco}$ queries and the final output contribute at most $(Q_{\mathrm{PCO}} + 1)/|\mathcal{K}|$.

Across the two cases, each $\textsc{Pco}$ query tests only one additional key accepted in $\mathbf{G}_4$. Thus blind key guesses contribute at most $(Q_{\mathrm{PCO}} + 1)/|\mathcal{K}|$ in total. By combining the above two cases, we have

$$\begin{aligned}&\left|\Pr\left[\mathbf{G}_3^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_4^{\mathcal{A}} \Rightarrow 1\right]\right| \\ &\qquad \leq \frac{Q_{\mathrm{PCO}} + Q_{\mathsf{H}'} + 1}{2^l} + (Q_{\mathrm{PK}} + N')(N' + Q_{\mathsf{G}}) \cdot \delta + \frac{Q_{\mathrm{PCO}} + 1}{|\mathcal{K}|}.\end{aligned}$$

Game $\mathbf{G}_5$, $\mathbf{G}_6$, and $\mathbf{G}_7$ (Decouple messages and randomness). We use the game sequence $\mathbf{G}_5 - \mathbf{G}_7$ to decouple the message m and randomness used in generating challenge ciphertext. Namely, in $\mathbf{G}_4$, challenge ciphertexts are generated as $\mathsf{c} := \mathsf{Enc}(\mathsf{pk}, m; \mathsf{G}(\alpha, m))$, while in $\mathbf{G}_7$, challenge ciphertexts are generated as $\mathsf{c} \leftarrow \mathsf{Enc}(\mathsf{pk}, m)$. These games are intermediate steps for preparing the reduction to the IND-CPA security, since in $\mathsf{IND\text{-}CPA}$, we cannot control the randomness of challenge ciphertexts.

Game $\mathbf{G}_5$: If $\mathcal{A}$ queries G on (α, m) such that $u \neq \bot$ and $\alpha = h_{\mathtt{tag}}(u)$, then we return $f_{\mathtt{msg}}(u, i)$, where $f_{\mathtt{msg}}$ is a random function and $(u, i) := h_{\mathtt{msg}}^{-1}(m)$ (cf. lines 47 to 49). Moreover, $f_{\mathtt{msg}}(u, i)$ is used as the randomness for the query to $\textsc{ChalCt}(u, i)$ (cf. line 28).

We claim that this change does not affect the winning advantage of $\mathcal{A}$. Firstly, both $h_{\mathtt{msg}}$ and $h_{\mathtt{tag}}$ are random injective functions, so the output distribution of G does not change. To see this, one can consider that, in $\mathbf{G}_4$, if $(u, i) := h_{\mathtt{msg}}^{-1}(m)$ and $\alpha = h_{\mathtt{tag}}(u)$, then $\mathsf{G}(\alpha, m) = f_{\mathsf{G}}(\alpha, m) = f_{\mathsf{G}}(h_{\mathtt{tag}}(u), h_{\mathtt{msg}}(u, i))$ uniquely determines (u, i), and thus determines $f_{\mathtt{msg}}(u, i)$. Then, since $\mathsf{G}(\alpha_u, m) = f_{\mathtt{msg}}(u, i)$ if $h_{\mathtt{msg}}^{-1}(m) = (u, i) \neq \bot$, we have $\mathsf{Enc}(\mathsf{pk}, m; \mathsf{G}(\alpha_u, m)) = \mathsf{Enc}(\mathsf{pk}, m; f_{\mathtt{msg}}(u, i))$. Therefore, the change made in $\mathbf{G}_5$ is conceptual and we have

$$\Pr\left[\mathbf{G}_4^{\mathcal{A}} \Rightarrow 1\right] = \Pr\left[\mathbf{G}_5^{\mathcal{A}} \Rightarrow 1\right].$$

Game $\mathbf{G}_6$: If $\mathcal{A}$ queries G on $(\cdot, m)$ or H on $(\cdot, m, \cdot)$, where $h_{\mathtt{msg}}^{-1}(m) \neq \bot$, then we abort the game and return 1 to indicate that the adversary wins this game (cf. line 50 and lines 55 to 56). This change only increases the winning advantage (i.e., makes the game output 1) of $\mathcal{A}$, thus we have

$$\Pr\left[\mathbf{G}_5^{\mathcal{A}} \Rightarrow 1\right] \leq \Pr\left[\mathbf{G}_6^{\mathcal{A}} \Rightarrow 1\right].$$

In $\mathbf{G}_6$, the random function $f_{\mathtt{msg}}$ will not be accessed via oracle G anymore and is only used in $\textsc{ChalCt}$ where it provides the encryption randomness, which is essential for the next step.

Game $\mathbf{G}_7$: The challenge ciphertext in query $\textsc{ChalCt}(u, i)$ is generated using the message $0^{|m|}$ instead of m, where $m = h_{\mathtt{msg}}(u, i)$.

We construct a reduction from the IND-CPA security of PKE. Reduction $\mathcal{B}$ is shown in Figure 16. If $\mathcal{B}$ plays $\mathsf{IND\text{-}CPA}_{\mathsf{PKE},0}$ (resp., $\mathsf{IND\text{-}CPA}_{\mathsf{PKE},1}$), then the challenge ciphertext is the encryption of the challenge message m (resp., $0^{|m|}$), and thus $\mathcal{B}$ perfectly simulates $\mathbf{G}_6$ (resp. $\mathbf{G}_7$) for $\mathcal{A}$. Therefore, we have

$$\left|\Pr\left[\mathbf{G}_6^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_7^{\mathcal{A}} \Rightarrow 1\right]\right| \leq \mathbf{Adv}_{\mathsf{PKE}}^{\mathsf{IND\text{-}CPA}}(\mathcal{B}),$$

Game $\mathcal{B}^{\text{ChalPk}',\text{ChalCt}'}(\mathsf{par}_{\mathsf{PKE}})$

$f_{\mathtt{kg}} \leftarrow \mathsf{Func}(\mathcal{U}, \mathcal{R}_{\mathtt{ekg}})$
$f_{\mathtt{prfk}} \leftarrow \mathsf{Func}(\mathcal{U}, \{0,1\}^l)$
$f_{\mathtt{G}} \leftarrow \mathsf{Func}(\{0,1\}^\ell \times \mathcal{M}, \mathcal{R})$
$f_{\mathtt{H}} \leftarrow \mathsf{Func}(\{0,1\}^\ell \times \mathcal{M} \times \mathcal{C}, \mathcal{K})$
$f_{\mathtt{h}} \leftarrow \mathsf{Func}(\{0,1\}^\ell \times \{0,1\}^l \times \mathcal{C}, \mathcal{K})$
$(h_{\mathtt{msg}}, h_{\mathtt{msg}}^{-1}) \leftarrow \mathsf{InjF}(\mathcal{U} \times \mathcal{I}, \mathcal{M})$
$(h_{\mathtt{tag}}, h_{\mathtt{tag}}^{-1}) \leftarrow \mathsf{InjF}(\mathcal{U}, \{0,1\}^\ell)$
$f_{\mathtt{Hkey}} \leftarrow \mathsf{Func}(\mathcal{U} \times \mathcal{C}, \mathcal{K})$
$\mathsf{par} := \mathsf{par}_{\mathsf{PKE}}$
$O := (\text{ChalPk}, \text{ChalCt}, \text{Pco})$
$(u^*, i^*, k^*) \leftarrow \mathcal{A}^O(\mathsf{par})$
$c^* := \text{ChalCt}(u^*, i^*)$
$b' := \text{Pco}(u^*, c^*, k^*)$
output b'

Oracle $\text{ChalPk}(u)$

$\mathsf{pk}_u := \text{ChalPk}'(u)$
$\alpha_u := h_{\mathtt{tag}}(u)$
$\mathsf{s}_u := f_{\mathtt{prfk}}(u)$
return $(\mathsf{pk}_u, \alpha_u)$

Oracle $\text{ChalCt}(u, i)$

$m := h_{\mathtt{msg}}(u, i)$
$c \leftarrow \text{ChalCt}'(u, i, m)$
$k := f_{\mathtt{Hkey}}(u, c)$
return c

Oracle $\text{Pco}(u, c, k)$

$\text{ChalPk}'(u)$
$\alpha_u := h_{\mathtt{tag}}(u)$
$\mathsf{s}_u := f_{\mathtt{prfk}}(u)$
$k'_0 := f_{\mathtt{Hkey}}(u, c)$
$k'_1 := \mathsf{H}'(\alpha_u, \mathsf{s}_u, c)$
if $k = k'_0 \vee k = k'_1$
  return 1
else return 0

Oracle $\mathsf{G}(\alpha, m)$

if $\bot \neq (u, i) := h_{\mathtt{msg}}^{-1}(m)$
  $\wedge\ \alpha = h_{\mathtt{tag}}(u)$
  abort and **output** 1
return $f_{\mathtt{G}}(\alpha, m)$

Oracle $\mathsf{H}(\alpha, m, c)$

if $h_{\mathtt{msg}}^{-1}(m) \neq \bot$
  abort and **output** 1
if $\bot \neq u := h_{\mathtt{tag}}^{-1}(\alpha)$
  if $m \neq \bot$
    $r' := \mathsf{G}(\alpha, m)$
    $\mathsf{pk}_u := \text{ChalPk}'(u)$
    if $c = \mathsf{Enc}(\mathsf{pk}_u, m; r')$
      return $f_{\mathtt{Hkey}}(u, c)$
return $f_{\mathtt{H}}(\alpha, m, c)$

Oracle $\mathsf{H}'(\alpha, \mathsf{s}, c)$

return $f_{\mathtt{h}}(\alpha, \mathsf{s}, c)$

Fig. 16. Reduction $\mathcal{B}$ in bounding $\mathbf{G}_6$ and $\mathbf{G}_7$ in the proof of Theorem 3. $\mathcal{B}$ simulates $\mathbf{G}_6$ or $\mathbf{G}_7$ for $\mathcal{A}$ to break the IND-CPA security of PKE.

and the running time and memory usage of $\mathcal{B}$ are

$$\mathbf{T}^*(\mathcal{B}) = \mathbf{T}^*(\mathcal{A}) + \mathrm{O}(Q) + \mathrm{O}(Q_{\mathsf{H}}) \cdot \mathbf{T}(\mathsf{PKE})$$
$$\mathbf{Mem}^*(\mathcal{B}) = \mathbf{Mem}^*(\mathcal{A}) + \log|\mathcal{U}| + \log|\mathcal{I}| + \mathrm{O}(\mathbf{Mem}(\mathsf{PKE})),$$

where Q is the total number of queries issued by $\mathcal{A}$.

Analysis of $\Pr\left[\mathbf{G}_7^{\mathcal{A}} \Rightarrow 1\right]$. We bound the winning probability of $\mathcal{A}$ in the final game $\mathbf{G}_7$. For convenience, we restate $\mathbf{G}_7$ in Figure 17. In $\mathbf{G}_7$, there are three ways for $\mathcal{A}$ to win the game:

- Case 1: $\mathcal{A}$ triggers the abort conditions in G and H. That is, $\mathcal{A}$ queries G on $(\cdot, m)$ or H on $(\cdot, m, \cdot)$ where $h_{\mathtt{msg}}^{-1}(m) \neq \bot$.

  In $\mathbf{G}_7$, all challenge ciphertexts are encryptions of $0^{|m|}$ and all messages in $\mathcal{M}$ are fixed-length. By the change made in $\mathbf{G}_6$-$\mathbf{G}_7$, the random function $f_{\mathtt{msg}}$ is decoupled from all other oracles, and thus $c := \mathsf{Enc}(\mathsf{pk}_u, 0^{|m|}; f_{\mathtt{msg}}(u, i))$ is just equivalent to $c \leftarrow \mathsf{Enc}(\mathsf{pk}_u, 0^{|m|})$ which uses internal uniform randomness. Hence, the oracle ChalCt does not reveal any information about $h_{\mathtt{msg}}$ (and its inverse). Since $h_{\mathtt{msg}}$ is an injection and $\mathcal{A}$ queries G and H at most Q_{G} and Q_{H} times, respectively, the probability that $\mathcal{A}$ finds a message m such that $h_{\mathtt{msg}}^{-1}(m) = (u, i)$ where $u \in \mathcal{U}$ is $\frac{Q_{\mathtt{H}} \cdot |\mathcal{U}| \cdot |\mathcal{I}| + Q_{\mathtt{G}} \cdot |\mathcal{I}|}{|\mathcal{M}|}$.
- Case 2: $\mathcal{A}$ successfully finds a (u^*, i^*, k^*) such that $\text{Pco}(u^*, c^*, k^*) = 1$, where $c^* = \text{ChalCt}(u^*, i^*)$ and $k^* = f_{\mathtt{Hkey}}(u^*, c^*)$.

  To obtain $f_{\mathtt{Hkey}}(u^*, c^*)$ via H, a value $\mathsf{Enc}(\mathsf{pk}_{u^*}, m; \mathsf{G}(\alpha_{u^*}, m))$ must equal a challenge ciphertext. Before the abort in Case 1, there are at most $Q_{\mathsf{G}} + Q_{\mathsf{H}}$ fresh inputs to G, including those queried inside H, and at most $Q_{\mathrm{CT}} + 1$ distinct challenge ciphertexts, including the final output check. For each pair under the same public key, spreadness bounds the collision probability by $2^{-\gamma}$ when the later randomness is sampled, regardless of which is generated first. By a union bound, this case has probability at most $(Q_{\mathsf{G}} + Q_{\mathsf{H}})(Q_{\mathrm{CT}} + 1)2^{-\gamma} + (Q_{\mathrm{Pco}} + 1)/|\mathcal{K}|$, where the second term counts all Pco attempts and the final output that guess the key before it is obtained through a corresponding H query.
- Case 3: $\mathcal{A}$ successfully finds a (u^*, i^*, k^*) such that $\text{Pco}(u^*, c^*, k^*) = 1$, where $c^* = \text{ChalCt}(u^*, i^*)$ and $k^* = \mathsf{H}'(\alpha_{u^*}, \mathsf{s}_{u^*}, c^*)$.

  Each user has a unique α_u, so each H' query guesses the seed of only one user. Counting seed guesses gives a bound of $(1 + Q_{\mathrm{Pco}} + Q_{\mathsf{H}'})/2^l$. Without querying H' at the true seed and ciphertext, $\mathcal{A}$ may instead guess the output key. Counting all Pco attempts and the final output gives an additional $(Q_{\mathrm{Pco}} + 1)/|\mathcal{K}|$. Thus this case has probability at most $(1 + Q_{\mathrm{Pco}} + Q_{\mathsf{H}'})/2^l + (Q_{\mathrm{Pco}} + 1)/|\mathcal{K}|$.

Game $\mathbf{G}_7$

$f_{\mathtt{kg}} \leftarrow \mathsf{Func}(\mathcal{U}, \mathcal{R}_{\mathtt{ekg}})$
$f_{\mathtt{prfk}} \leftarrow \mathsf{Func}(\mathcal{U}, \{0,1\}^l)$
$f_{\mathtt{G}} \leftarrow \mathsf{Func}(\{0,1\}^\ell \times \mathcal{M}, \mathcal{R})$
$f_{\mathtt{H}} \leftarrow \mathsf{Func}(\{0,1\}^\ell \times \mathcal{M} \times \mathcal{C}, \mathcal{K})$
$f_{\mathtt{h}} \leftarrow \mathsf{Func}(\{0,1\}^\ell \times \{0,1\}^l \times \mathcal{C}, \mathcal{K})$
$(h_{\mathtt{msg}}, h_{\mathtt{msg}}^{-1}) \leftarrow \mathsf{InjF}(\mathcal{U} \times \mathcal{I}, \mathcal{M})$
$(h_{\mathtt{tag}}, h_{\mathtt{tag}}^{-1}) \leftarrow \mathsf{InjF}(\mathcal{U}, \{0,1\}^\ell)$
$f_{\mathtt{Hkey}} \leftarrow \mathsf{Func}(\mathcal{U} \times \mathcal{C}, \mathcal{K})$
$f_{\mathtt{msg}} \leftarrow \mathsf{Func}(\mathcal{U} \times \mathcal{I}, \mathcal{R})$
$\mathsf{par} \leftarrow \mathsf{Setup}$
$O := (\textsc{ChalPk}, \textsc{ChalCt}, \textsc{Pco})$
$(u^*, i^*, k^*) \leftarrow \mathcal{A}^O(\mathsf{par})$
return $\textsc{Pco}(u^*, \textsc{ChalCt}(u^*, i^*), k^*)$

Oracle $\textsc{ChalPk}(u)$

$(\mathsf{pk}_u, \mathsf{sk}_u) := \mathsf{PKE.KG}(; f_{\mathtt{kg}}(u))$
$\alpha_u := h_{\mathtt{tag}}(u)$, $\mathsf{s}_u := f_{\mathtt{prfk}}(u)$
return $(\mathsf{pk}_u, \alpha_u)$

Oracle $\textsc{ChalCt}(u, i)$

$\textsc{ChalPk}(u)$
$m := h_{\mathtt{msg}}(u, i)$
$c := \mathsf{Enc}(\mathsf{pk}_u, 0^{|m|}; f_{\mathtt{msg}}(u, i))$
$k := f_{\mathtt{Hkey}}(u, c)$
return c

Oracle $\textsc{Pco}(u, c, k)$

$\textsc{ChalPk}(u)$
$k'_0 = f_{\mathtt{Hkey}}(u, c)$
$k'_1 = \mathsf{H}'(\alpha_u, \mathsf{s}_u, c)$
if $k = k'_0 \ \vee \ k = k'_1$: **return** 1
return 0

Oracle $\mathsf{G}(\alpha, m)$

$(u, i) := h_{\mathtt{msg}}^{-1}(m)$
if $(u, i) \neq \bot \ \wedge \ \alpha = h_{\mathtt{tag}}(u)$
  abort and **output** 1
return $f_{\mathtt{G}}(\alpha, m)$

Oracle $\mathsf{H}(\alpha, m, c)$

if $h_{\mathtt{msg}}^{-1}(m) \neq \bot$
  abort and **output** 1
if $\bot \neq u := h_{\mathtt{tag}}^{-1}(\alpha)$
  $\textsc{ChalPk}(u)$
  if $m \neq \bot$
    $\wedge \ c = \mathsf{Enc}(\mathsf{pk}_u, m; \mathsf{G}(\alpha, m))$
    return $f_{\mathtt{Hkey}}(u, c)$
return $f_{\mathtt{H}}(\alpha, m, c)$

Oracle $\mathsf{H}'(\alpha, \mathsf{s}, c)$

return $f_{\mathtt{h}}(\alpha, \mathsf{s}, c)$

Fig. 17. Game $\mathbf{G}_7$ in the proof of Theorem 3. For simplicity, we assume that pk_u is generated from $(\mathsf{pk}_u, \mathsf{sk}_u) := \mathsf{PKE.KG}(; f_{\mathtt{kg}}(u))$, $\alpha_u := h_{\mathtt{tag}}(u)$, and $\mathsf{s}_u := f_{\mathtt{prfk}}(u)$ since $\mathbf{G}_1$.

By combining the probability bounds of the above three cases, we have

$$\Pr\left[\mathbf{G}_7^{\mathcal{A}} \Rightarrow 1\right] \leq \frac{Q_{\mathsf{H}} \cdot |\mathcal{U}| \cdot |\mathcal{I}| + Q_{\mathsf{G}} \cdot |\mathcal{I}|}{|\mathcal{M}|} + \frac{2(Q_{\mathrm{Pco}} + 1)}{|\mathcal{K}|} + \frac{(Q_{\mathsf{G}} + Q_{\mathsf{H}})(Q_{\mathrm{Ct}} + 1)}{2^\gamma} + \frac{1 + Q_{\mathrm{Pco}} + Q_{\mathsf{H}'}}{2^l}$$

Since $N' = Q_{\mathsf{H}} + Q_{\mathrm{Ct}} + Q_{\mathrm{Pco}} + 1$, the combined spreadness loss is at most

$$\left(\frac{(Q_{\mathsf{G}} + N')^2}{2} + (Q_{\mathsf{G}} + Q_{\mathsf{H}})(Q_{\mathrm{Ct}} + 1)\right) 2^{-\gamma} \leq (Q_{\mathsf{G}} + N')^2 2^{-\gamma}.$$

Together with the $(Q_{\mathrm{Pco}}+1)/|\mathcal{K}|$ term from $\mathbf{G}_3$ to $\mathbf{G}_4$, the total key-guessing loss is at most $3(Q_{\mathrm{Pco}}+1)/|\mathcal{K}|$. In conclusion, by combining all probability differences, we have

$$\begin{aligned}
&\mathbf{Adv}_{\mathsf{KEM}}^{\mathsf{OW\text{-}PCA}}(\mathcal{A}) = \Pr\left[\mathsf{OW\text{-}PCA}_{\mathsf{KEM}}^{\mathcal{A}} \Rightarrow 1\right] \\
&\leq \mathbf{Adv}_{\mathsf{PKE}}^{\mathsf{IND\text{-}CPA}}(\mathcal{B}) + (Q_{\mathrm{Pk}} + N')(N' + Q_{\mathsf{G}}) \cdot \delta + \frac{Q_{\mathsf{H}} + (Q_{\mathsf{G}} + N')^2}{2^\gamma} \\
&\quad + \frac{3(Q_{\mathrm{Pco}} + 1)}{|\mathcal{K}|} + \frac{Q_{\mathsf{H}} \cdot |\mathcal{U}| \cdot |\mathcal{I}| + Q_{\mathsf{G}} \cdot |\mathcal{I}| + (Q_{\mathrm{Ct}} + 1)^2}{|\mathcal{M}|} \\
&\quad + \frac{2(Q_{\mathrm{Pco}} + Q_{\mathsf{H}'} + 1)}{2^l} + \frac{(N' + Q_{\mathrm{Pk}})^2}{2^{\ell+1}}
\end{aligned}$$

This concludes the proof of Theorem 3.

From Single-user, Single Challenge Primitives. In Theorem 3, if we consider the single-user, single-challenge setting, then there is only one challenge public key and one challenge ciphertext, so we have Corollary 1 (by setting $|\mathcal{U}| := 1$, $|\mathcal{I}| := 1$, $Q_{\mathrm{Pk}} := 1$, and $Q_{\mathrm{Ct}} := 1$). In the single-user, single-challenge setting, we do not need to maintain the namespaces of users and challenge ciphertexts and the repeated

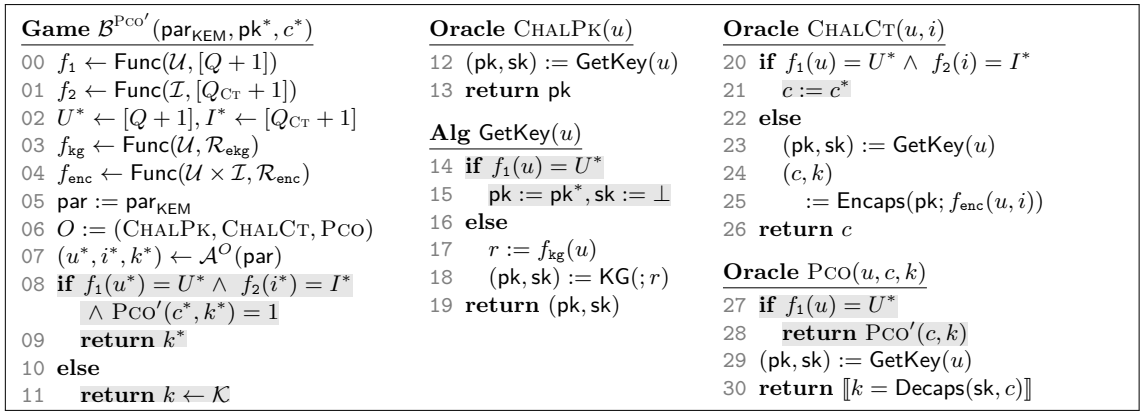

Fig. 18. Reduction $\mathcal{B}$ for proving Lemma 11.

queries to oracles ChalPk and ChalCt. The reduction reuses the same challenge ciphertext for the final output check. Corollary 1 shows that IND-CPA-secure PKE can be tightly transformed into a OW-PCA-secure KEM by the generic construction in Figure 14 in the single-user, single-challenge setting.

Corollary 1. *Let* KEM *be the KEM construction presented in Figure 14. If the PKE scheme* PKE *is* $(1-\delta)$*-correct and* γ*-spread, then for any adversary* $\mathcal{A}$ *against the OW-PCA security of* KEM *in the single-user, single-challenge setting, there exists an adversary* $\mathcal{B}$ *against the IND-CPA security of* PKE *in the single-user, single-challenge setting such that*

$$\begin{aligned}
\mathbf{Adv}^{\mathsf{OW\text{-}PCA}}_{\mathsf{KEM}}(\mathcal{A}) &\le \mathbf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{PKE}}(\mathcal{B}) + (1+N')(N'+Q_{\mathsf{G}})\cdot\delta + \frac{Q_{\mathsf{H}} + (Q_{\mathsf{G}}+N')^2}{2^{\gamma}} \\
&\quad + \frac{3(Q_{\mathrm{Pco}}+1)}{|\mathcal{K}|} + \frac{Q_{\mathsf{G}}+Q_{\mathsf{H}}+4}{|\mathcal{M}|} \\
&\quad + \frac{2(Q_{\mathrm{Pco}}+Q_{\mathsf{H}'}+1)}{2^{l}} + \frac{(N'+1)^2}{2^{\ell+1}} \\
\mathbf{T}^*(\mathcal{B}) &= \mathbf{T}^*(\mathcal{A}) + \mathrm{O}(Q)\cdot\mathbf{T}(\mathsf{PKE}) \\
\mathbf{Mem}^*(\mathcal{B}) &= \mathbf{Mem}^*(\mathcal{A}) + \mathrm{O}(\mathbf{Mem}(\mathsf{PKE})),
\end{aligned}$$

where $\mathbf{T}(\mathsf{PKE})$ *and* $\mathbf{Mem}(\mathsf{PKE})$ *are the running time and memory usage of one execution of* PKE*, respectively, and* $\mathcal{A}$ *issues at most* $Q_{\mathrm{Pco}}, Q_{\mathsf{G}}, Q_{\mathsf{H}}$*, and* $Q_{\mathsf{H}'}$ *queries to oracles* Pco,G, H*, and* H′*, respectively.* Q *is the total number of queries issued by* $\mathcal{A}$*, and* $N' := Q_{\mathsf{H}} + Q_{\mathrm{Pco}} + 2$*.*

In Lemma 11, we further show that OW-PCA security in the single-user, single-challenge setting implies the same security in the multi-user, multi-challenge setting. Although the reduction is not tight in terms of the winning advantage, it preserves the memory-tightness. Since the proof of Lemma 11 is quite simple, we only give the construction of the reduction in Figure 18.

Lemma 11. *For any adversary* $\mathcal{A}$ *against the* OW-PCA *security of* KEM *in the multi-user, multi-challenge setting (denoted as* OW-mucPCA*), there exists an adversary* $\mathcal{B}$ *(cf. Figure 18) against the* OW-PCA *security of* KEM *in the single-user, single-challenge setting (denoted as* OW-sucPCA*) such that*

$$\begin{aligned}
\mathbf{Adv}^{\mathsf{OW\text{-}mucPCA}}_{\mathsf{KEM}}(\mathcal{A}) &\le e^2\cdot(Q+1)(Q_{\mathrm{CT}}+1)\cdot\mathbf{Adv}^{\mathsf{OW\text{-}sucPCA}}_{\mathsf{KEM}}(\mathcal{B}) \\
\mathbf{T}^*(\mathcal{B}) &\approx \mathbf{T}^*(\mathcal{A}) + \mathrm{O}(Q)\cdot\mathbf{T}(\mathsf{KEM}) \\
\mathbf{Mem}^*(\mathcal{B}) &\approx \mathbf{Mem}^*(\mathcal{A}) + \log|\mathcal{U}| + \log|\mathcal{I}| + \mathrm{O}(\mathbf{Mem}(\mathsf{KEM})),
\end{aligned}$$

where $\mathbf{T}^*(\mathsf{KEM})$ *and* $\mathbf{Mem}^*(\mathsf{KEM})$ *are the (reduced) running time and memory usage of one execution of* KEM*, respectively.* Q_{PK}*,* Q_{CT}*, and* Q_{Pco} *are fixed upper bounds on the numbers of direct queries made by* $\mathcal{A}$ *to oracles* ChalPk, ChalCt*, and* Pco*, respectively.* $Q = Q_{\mathrm{PK}} + Q_{\mathrm{Pco}} + Q_{\mathrm{CT}}$ *bounds the total number of these queries. Including the final output, at most* $Q+1$ *distinct users and* $Q_{\mathrm{CT}}+1$ *distinct challenge indices occur.*

F Deferred Proofs of Lemmas in Section 4

F.1 Proof of Lemma 2

In $\mathbf{G}_1$, all randomness used in $\mathsf{SIG.KG}$ is generated by random function $f_{\mathtt{skg}}$ (cf. line 22), and the randomness used in $\mathsf{SIG.Sign}$ in $\textsc{Init}_{\mathrm{I}}$ and $\textsc{Der}_{\mathrm{R}}$ is generated by random functions $f_{\mathtt{sI}}$ and $f_{\mathtt{sR}}$, respectively (cf. lines 31 and 54). For KEM, we first set up a global counter $\mathsf{cnt} \in [N_{\mathtt{ses}}]$ which is initialized to 0 and incremented by 1 every time a query to $\textsc{Init}_{\mathrm{I}}$ or $\textsc{Der}_{\mathrm{R}}$ is made (cf. lines 24 and 43). We then use this counter to compute the ephemeral KEM key pairs from $\mathsf{KEM.KG}$ and encapsulated ciphertexts from $\mathsf{KEM.Encaps}$ using random functions $f_{\mathtt{ekg}}$ and $f_{\mathtt{enc}}$, respectively (cf. line 26 and lines 44 to 45). We will discuss how to choose the upper bound N later.

Since we use random functions to generate randomness, the only chance for the adversary to detect this change is that the same input is given to these random functions twice. We consider three cases:

- For the randomness to $\mathsf{KEM.KG}$ and $\mathsf{KEM.Encaps}$ (cf. lines 26 and 44), the input to the random functions is just the counter cnt, which is unique for each query, so no collision happens and this change is perfectly indistinguishable.
- For the randomness to $\mathsf{SIG.KG}$ (cf. line 22), the input is the user identifier, which is also unique for each user, so no collision happens and this change is also perfectly indistinguishable.
- For the randomness to $\mathsf{SIG.Sign}$ (cf. lines 31 and 54), the input is the message to be signed. These inputs are (u, α, epk) for $f_{\mathtt{sI}}$ and $(u, v, \alpha, \beta, epk, c, \sigma_1)$ for $f_{\mathtt{sR}}$. Including the identities and σ_1 cannot create additional input collisions, so the nonce-based birthday bound below is unchanged. For each session, the protocol message output by it includes a nonce α or β, which is generated uniformly at random from $\{0,1\}^{\ell}$ and included in the message to be signed. Therefore, the probability that two different sessions have the same message to be signed is at most $(N_{\mathtt{ses}})^2/2^{\ell+1}$ (by the birthday bound over $N_{\mathtt{ses}}$ queries to $\textsc{Init}_{\mathrm{I}}$ and $\textsc{Der}_{\mathrm{R}}$).

Therefore, we have the following bound

$$\left|\Pr\left[\mathbf{G}_0^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_1^{\mathcal{A}} \Rightarrow 1\right]\right| \leq \frac{N_{\mathtt{ses}}^2}{2^{\ell+1}},$$

which concludes the proof of Lemma 2.

F.2 Proof of Lemma 3

In $\mathbf{G}_2$, the change is as follows:

- On a session initialization query to $\textsc{Init}_{\mathrm{I}}(u, v)$, the first sequence number seq_1 of an initiator session is generated by $f'_{\mathtt{seq}}(u, v, \mathsf{cnt})$, where cnt is the global counter which increments on each query to $\textsc{Init}_{\mathrm{I}}$ or $\textsc{Der}_{\mathrm{R}}$ (cf. line 34).
- On a responder session completion query to $\textsc{Der}_{\mathrm{R}}(u, v, \mathsf{M}_u)$, the sequence number seq of the session is generated by $f_{\mathtt{seq}}(u, v, \mathsf{M}_u, \mathsf{M}_v)$, where M_v is the response message generated in the same query.
- On an initiator session completion query to $\textsc{Der}_{\mathrm{I}}(u, v, \mathsf{seq}_1, \mathsf{M}_v)$, the second sequence number seq_2 of the session is generated by $f_{\mathtt{seq}}(\mathsf{seq}_1, u, v, \mathsf{M}_u, \mathsf{M}_v)$ (cf. line 77), where M_u is the initiation message stored in the session state.

This change can be decomposed into two steps. First, we replace uniform sampling without replacement (i.e., sampling from $\mathcal{SEQ} \backslash \mathcal{S}_{\mathtt{seq}}$ in a non-repeating way) with uniform sampling with replacement (i.e., sampling from $\mathcal{SEQ}$ in a possibly-repeating way, without $\mathcal{S}_{\mathtt{seq}}$). Second, we replace uniform sampling with replacement by the use of a random function. For convenience, we combine these two steps into a single game transition.

We first argue that replacing sampling without replacement by sampling with replacement is indistinguishable to the adversary except with small probability. Indeed, the two experiments proceed identically until a collision occurs among the generated sequence numbers. Thus, the only way for the adversary to detect this change is if a freshly generated sequence number collides with a previously generated one. By applying a standard birthday bound (where the size of the sample set $\mathcal{S}_{\mathtt{seq}}$ is at most $N_{\mathtt{ses}}$), this event occurs with probability at most $N_{\mathtt{ses}}^2/2|\mathcal{SEQ}|$ (by the birthday bound).

We then show that sampling with replacement can be replaced by the use of a random function. The only way for the adversary to detect this change is if the same input is given to the random function twice. More specifically,

– For $\mathsf{seq}_1 = f'_{\mathsf{seq}}(u, v, \mathsf{cnt})$, the counter cnt is unique for each query, so no input collision happens.
– For $\mathsf{seq} = f_{\mathsf{seq}}(u, v, \mathsf{M}_u, \mathsf{M}_v)$, the inputs depend on the protocol messages. Since two different responder sessions have different nonces β's with high probability (by the nonce collision probability $2^{-\ell}$), by a union bound over all responder sessions, the probability that two different responder sessions have the same inputs to f_{seq} is at most $N_{\mathsf{ses}}^2/2^{\ell+1}$.
– For $\mathsf{seq}_2 = f_{\mathsf{seq}}(\mathsf{seq}_1, u, v, \mathsf{M}_u, \mathsf{M}_v)$, the analysis is somehow subtler because, in our model, an initiator session allows repeated completion queries. We first need to ensure that the same completion queries of the same initiator session will produce the same seq_2. This is indeed the case because the inputs to f_{seq} are all fixed for the same completion query.

  Next, we argue that different initiator sessions will not collide on the inputs to f_{seq} except with small probability. This is ensured by the fact that different initiator sessions either have different seq_1's or different initiator messages M_u's with high probability (by the nonce α sampled from $\{0,1\}^\ell$ uniformly at random). Combining these two cases, the probability that two different initiator sessions have the same inputs to f_{seq} is at most $N_{\mathsf{ses}}^2/2|\mathcal{SEQ}| + N_{\mathsf{ses}}^2/2^{\ell+1}$.

Therefore, by combining the above two steps, we have the following bound

$$\left|\Pr\left[\mathbf{G}_1^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_2^{\mathcal{A}} \Rightarrow 1\right]\right| \leq \frac{N_{\mathsf{ses}}^2}{|\mathcal{SEQ}|} + \frac{N_{\mathsf{ses}}^2}{2^\ell},$$

which concludes the proof of Lemma 3.

F.3 Proof of Lemma 4

In $\mathbf{G}_3$ and $\mathbf{G}_4$, an seq_1 output from $\textsc{Init}_{\mathrm{I}}(u, v)$ is bound to the unique tuple (u, v, cnt) by h_{seq} for some cnt. When $\mathcal{A}$ queries $\textsc{Der}_{\mathrm{I}}(u', v', \mathsf{seq}_1, *)$, the oracle (in both games) will check if $(u', v') = (u, v)$ and recover the unique cnt. Hence, replaying seq_1 to a different query $\textsc{Der}_{\mathrm{I}}(u', v', *)$ does not help. Moreover, since the initiator session state is determined by (u, v, cnt) in both games, the query $\textsc{Der}_{\mathrm{I}}(u, v, *)$ can always restore the correct session state from $\textsc{Init}_{\mathrm{I}}(u, v)$. Hence, replaying seq_1 from $\textsc{Init}_{\mathrm{I}}$ does not help to distinguish $\mathbf{G}_3$ from $\mathbf{G}_4$.

Therefore, to distinguish $\mathbf{G}_4$ from $\mathbf{G}_3$, $\mathcal{A}$ has to query $\textsc{Der}_{\mathrm{I}}(u, v, \mathsf{seq}_1, *)$, where seq_1 is not from any oracle, such that $(u', v', \mathsf{cnt}') = h_{\mathsf{seq}}^{-1}(\mathsf{seq}_1)$, $(u, v) = (u', v')$, and $\mathsf{cnt}' \leq \mathsf{cnt}$. Since h_{seq} is a random injection, $h_{\mathsf{seq}}(\mathcal{U}^2 \times [N_{\mathsf{ses}}])$ is a random subset of $\mathcal{SEQ}$, and given user identities u and v, the number of points in $\mathcal{SEQ}$ that can be inverted to (u, v, cnt') by h_{seq}^{-1} for some $\mathsf{cnt}' \leq \mathsf{cnt} \leq N_{\mathsf{ses}}$ is at most N_{ses}. Moreover, since $\mathcal{A}$ learns at most N_{ses} points from the $\textsc{Init}_{\mathrm{I}}$ oracle, the probability that an arbitrary seq_1 (that is not from any oracle) satisfies $(u, v, \mathsf{cnt}') = h_{\mathsf{seq}}^{-1}(\mathsf{seq}_1)$, $(u, v) = (u', v')$, and $\mathsf{cnt}' \leq N_{\mathsf{ses}}$ is $\frac{N_{\mathsf{ses}}}{|\mathcal{SEQ}| - N_{\mathsf{ses}}}$ (uniformly random over the unused image points of h_{seq}). If $N_{\mathsf{ses}} = 0$, there are no completion queries and the claim is immediate. If $|\mathcal{SEQ}| < 2N_{\mathsf{ses}}$, the claimed upper bound $2N_{\mathsf{ses}}^2/|\mathcal{SEQ}|$ is at least 1, so the claim is again immediate. Otherwise $|\mathcal{SEQ}| \geq 2N_{\mathsf{ses}}$, and we have

$$\frac{N_{\mathsf{ses}}}{|\mathcal{SEQ}| - N_{\mathsf{ses}}} \leq \frac{N_{\mathsf{ses}}}{|\mathcal{SEQ}|/2} = \frac{2N_{\mathsf{ses}}}{|\mathcal{SEQ}|}$$

As $\mathcal{A}$ queries $\textsc{Der}_{\mathrm{I}}$ at most N_{ses} times, by a union bound, we have

$$\left|\Pr\left[\mathbf{G}_3^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_4^{\mathcal{A}} \Rightarrow 1\right]\right| \leq \frac{2N_{\mathsf{ses}}^2}{|\mathcal{SEQ}|},$$

which concludes the proof of Lemma 4.

F.4 Proof of Lemma 5

Let f_{k} be the random function introduced in $\mathbf{G}_6$. In this game, the random oracle H is modified so that session keys can be computed without using the underlying KEM keys. More specifically, on a query $\mathsf{H}(\mathsf{pk}_1, \mathsf{pk}_2, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k)$, we initialise $k^* := k$ and $d := 0$, and recover

$$(b_1, u_1, v_1, \mathsf{cnt}'_1, \mathsf{cnt}_1) := h_{\mathsf{nce}}^{-1}(\alpha) \quad \text{and} \quad (b_2, u_2, v_2, \mathsf{cnt}'_2, \mathsf{cnt}_2) := h_{\mathsf{nce}}^{-1}(\beta).$$

The flag d records whether the initiator nonce α allows us to reconstruct a key pair whose public key matches the epk from the query. We then proceed as follows:

– If $h_{\mathtt{nce}}^{-1}(\alpha) \neq \bot$ and $b_1 = 0$, we reconstruct $(epk', esk') := \mathsf{KEM.KG}(; f_{\mathtt{ekg}}(\mathsf{cnt}_1))$. If $epk = epk'$, we set $d := 1$ and compute $k' := \mathsf{KEM.Decaps}(esk', c)$. If $k = k'$, then we set $k^* := f_{\mathtt{k}}(epk, c)$ (cf. lines 80 to 86).
– If $d = 0$, $h_{\mathtt{nce}}^{-1}(\beta) \neq \bot$, and $b_2 = 1$, we reconstruct $(c', k') := \mathsf{KEM.Encaps}(epk; f_{\mathtt{enc}}(epk, \mathsf{cnt}_2))$. If $(c, k) = (c', k')$, we set $k^* := f_{\mathtt{k}}(epk, c)$ (cf. lines 87 to 89).
– Finally, we return $f_{\mathtt{H}}(\mathsf{pk}_1, \mathsf{pk}_2, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k^*)$.

In particular, once a matching key pair has been reconstructed, d remains 1 even if k fails the decapsulation check. Thus, the responder branch is used only when the initiator branch cannot recover a matching key pair.

Essentially, these changes replace the KEM key recognised by the applicable branch with $f_{\mathtt{k}}(epk, c)$. We also modify the computation of session keys to make it consistent with the modified oracle H:

– The outputs of Rev and the real keys returned by Test are computed by the internal procedure ComputeSK (cf. Figure 8). Both oracles first use CheckSid to check whether the queried session identifier exists. For a session identifier of the form

$$\mathsf{sID} = (*, u, v, (\alpha, epk, \sigma_1), (\beta, c, \sigma_2)),$$

the procedure ComputeSK (cf. lines 17 and 25) returns

$$f_{\mathtt{H}}(\mathsf{pk}_u, \mathsf{pk}_v, \alpha, \beta, epk, c, \sigma_1, \sigma_2, f_{\mathtt{k}}(epk, c)).$$

– For consistency, we compute the session keys in $\textsc{Der}_{\mathrm{I}}$ and $\textsc{Der}_{\mathrm{R}}$ using ComputeSK as well (cf. lines 56 and 74). These computations can be omitted because session keys no longer need to be stored. All session keys can be recomputed by ComputeSK whenever needed. We include them only to make the game transition explicit.

We first analyse the distribution of the modified random oracle H. Let

$$\mathsf{Trans} := (\mathsf{pk}_1, \mathsf{pk}_2, \alpha, \beta, epk, c, \sigma_1, \sigma_2).$$

For each fixed Trans, determine d as described above. If $d = 1$, let esk' be the reconstructed secret key. If $d = 0$, $h_{\mathtt{nce}}^{-1}(\beta) \neq \bot$, and $b_2 = 1$, let

$$(c', k') := \mathsf{KEM.Encaps}(epk; f_{\mathtt{enc}}(epk, \mathsf{cnt}_2)).$$

Then we can write $\mathsf{H}(\mathsf{Trans}, k) = f_{\mathtt{H}}(f(\mathsf{Trans}, k))$, where

$$f(\mathsf{Trans}, k) = \begin{cases} (\mathsf{Trans}, f_{\mathtt{k}}(epk, c)), & \text{if } d = 1 \ \wedge \ k = \mathsf{KEM.Decaps}(esk', c), \\ (\mathsf{Trans}, f_{\mathtt{k}}(epk, c)), & \text{if } \begin{cases} d = 0 \ \wedge \ h_{\mathtt{nce}}^{-1}(\beta) \neq \bot \ \wedge \ b_2 = 1 \\ \wedge \ (c, k) = (c', k'), \end{cases} \\ (\mathsf{Trans}, k), & \text{otherwise.} \end{cases}$$

Importantly, d depends on Trans but not on k. Consequently, for each fixed Trans, f replaces at most one KEM key with $f_{\mathtt{k}}(epk, c)$ and leaves all other keys unchanged. Thus, two distinct queries can produce the same input to $f_{\mathtt{H}}$ only if they have the same Trans and an unchanged key equals $f_{\mathtt{k}}(epk, c)$. Since $f_{\mathtt{k}}$ is a random function, such an equality occurs only with small probability. There are at most $Q_{\mathtt{H}}$ adversarial queries and $N_{\mathtt{ses}}$ session-key inputs. By a union bound over the relevant pairs, the probability of such a collision is at most

$$\frac{(Q_{\mathtt{H}} + N_{\mathtt{ses}})^2}{|\mathcal{K}|}.$$

Unless such a collision occurs, distinct queries are answered using distinct inputs to $f_{\mathtt{H}}$. Hence, the modified H behaves as a random oracle in $\mathcal{A}$'s view.

Next, we show that the modified session-key computations are consistent with this oracle, except when a relevant KEM decryption error occurs.

1. Consider an established initiator session. Its nonce α allows the oracle to reconstruct the same key pair used by the session, so $d = 1$. The KEM key used by the initiator is $\mathsf{KEM.Decaps}(esk', c)$ and is therefore replaced by $f_{\mathtt{k}}(epk, c)$. Thus, its session key agrees with the output of $\mathsf{ComputeSK}$.
2. Now consider an established responder session. If $d = 0$, its nonce β allows the oracle to reconstruct the same encapsulation using $f_{\mathtt{enc}}(epk, \mathsf{cnt}_2)$. The responder's encapsulated key is therefore replaced by $f_{\mathtt{k}}(epk, c)$.
3. However, if $d = 1$, then the oracle uses the reconstructed secret key esk' to check the responder's encapsulated key. The replacement is consistent provided that

$$(c, k) := \mathsf{KEM.Encaps}(epk; f_{\mathtt{enc}}(epk, \mathsf{cnt}_2)) \quad \Longrightarrow \quad k = \mathsf{KEM.Decaps}(esk', c),$$

where $(epk, esk') = \mathsf{KEM.KG}(; f_{\mathtt{ekg}}(\mathsf{cnt}_1))$. Hence, in the absence of these decryption errors, the session keys computed by both $\textsc{Der}_{\mathrm{I}}$ and $\textsc{Der}_{\mathrm{R}}$ agree with $\mathsf{ComputeSK}$.

Then, we bound the probability of a relevant decryption error. Consider all counter-indexed key pairs $(epk_i, esk_i) := \mathsf{KEM.KG}(; f_{\mathtt{ekg}}(i))$ and, for each such key pair, the encapsulations

$$(c_{i,j}, k_{i,j}) := \mathsf{KEM.Encaps}(epk_i; f_{\mathtt{enc}}(epk_i, j)), \qquad i, j \in [N_{\mathtt{ses}}].$$

These pairs cover all possible reconstructions. For each fixed (i, j), both key generation and encapsulation use independent uniform randomness (as the KEM correctness game requires, cf. Definition 1). Since KEM is $(1 - \delta_{\mathsf{KEM}})$-correct,

$$\Pr[k_{i,j} \neq \mathsf{KEM.Decaps}(esk_i, c_{i,j})] \leq \delta_{\mathsf{KEM}}.$$

A union bound over at most $N_{\mathtt{ses}}^2$ index pairs therefore bounds the probability of any relevant decryption error by $N_{\mathtt{ses}}^2 \delta_{\mathsf{KEM}}$.

Finally, for an existing session identifier, $\textsc{Rev}$ and $\textsc{Test}$ use the same procedure $\mathsf{ComputeSK}$ to obtain the real session key. Their outputs are therefore consistent with the session-key computations above. By the abort convention in $\mathbf{G}_5$, it suffices to consider session identifiers that have already been output by $\textsc{Der}_{\mathrm{I}}$ or $\textsc{Der}_{\mathrm{R}}$.

Combining the collision and decryption-error bounds, we obtain

$$\left|\Pr\left[\mathbf{G}_5^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_6^{\mathcal{A}} \Rightarrow 1\right]\right| \leq \frac{(Q_{\mathtt{H}} + N_{\mathtt{ses}})^2}{|\mathcal{K}|} + N_{\mathtt{ses}}^2 \cdot \delta_{\mathsf{KEM}},$$

which concludes the proof of Lemma 5.

F.5 Proof of Lemma 7

In this section, we prove that, in $\mathbf{G}_8$, $(u, v, \mathsf{M}_u, \mathsf{M}_v) \in \mathcal{S}_{\mathtt{msg}} \iff \mathsf{CheckMsgR}(u, v, \mathsf{M}_u, \mathsf{M}_v) = 1$. As it is straightforward to show $\mathsf{CheckMsgR}$ never creates false-negatives, we only need to show the non-trivial direction that if $\mathsf{CheckMsgR}(u, v, \mathsf{M}_u, \mathsf{M}_v) = 1$, then $(u, v, \mathsf{M}_u, \mathsf{M}_v) \in \mathcal{S}_{\mathtt{msg}}$ except with some negligible probability.

Since $\mathsf{CheckMsgR}$ calls $\mathsf{CheckMsgI}$ to check if M_u is a valid initiator message from u (i.e., $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$), we first include the probability of false positives arising from $\mathsf{CheckMsgI}$. Since there are at most $N_{\mathtt{ses}}$ test sessions, $\mathsf{CheckMsgR}$ runs at most $N_{\mathtt{ses}}$ times, and each time $\mathsf{CheckMsgI}$ runs once, by applying the analysis of Lemma 6 and a union bound, the probability that $\mathsf{CheckMsgR}$ produces false positives from $\mathsf{CheckMsgI}$ is at most $\frac{2N_{\mathtt{ses}}^2|\mathcal{U}|}{2^\ell}$. By ensuring $\mathsf{CheckMsgI}(u, \mathsf{M}_u) = 1$, we have $\perp \neq (*, \mathsf{cnt}') = h_{\mathtt{nce}}^{-1}(\alpha)$ and $\mathsf{cnt}' \neq 0$.

Then, we only need to consider the case that $\mathsf{CheckMsgI}$ does not produce false positives. Similar to the analysis of $\mathsf{CheckMsgI}$, given $(u, v, \mathsf{M}_u, \mathsf{M}_v)$, we consider the following cases when $\mathsf{CheckMsgR}(u, v, \mathsf{M}_u, \mathsf{M}_v)$ returns 1:

Case 1: β in M_v is forged by $\mathcal{A}$. In this case, β is not generated from the oracles in the game. This case is similar to the first case of $\mathsf{CheckMsgI}$, except that here we need to consider if β_v matches the α in M_u. By the code of $\mathsf{CheckMsgR}$ (cf. Figure 8), to make $h_{\mathtt{nce}}^{-1}(\beta)$ satisfy the if-condition, the inverse of the given β should first fall into the set

$$\left\{(1, u, v, \mathsf{cnt}', \mathsf{cnt}) \;\middle|\; \mathsf{cnt} \in [N_{\mathtt{ses}}] \wedge (*, \mathsf{cnt}') = h_{\mathtt{nce}}^{-1}(\alpha)\right\},$$

which has $N_{\mathtt{ses}}$ points. Since $h_{\mathtt{nce}}$ is a random injective function and $\mathcal{A}$ has seen at most $N_{\mathtt{ses}}$ points of the image of $h_{\mathtt{nce}}$, the probability that $\mathcal{A}$ successfully forges such β in $N_{\mathtt{ses}}$ queries is bounded by $\frac{2N_{\mathtt{ses}}^2}{2^\ell}$. Note that here we do not need the $|\mathcal{U}|$ factor in the probability bound as in the case of CheckMsgI, because here we fix both the sender and receiver identities in the set.

Case 2: β in M_v is from some $\textsc{Der}_{\mathrm{R}}$ query. Let $(u_0, v_0, \mathsf{M}_0, \mathsf{M}'_0) \in \mathcal{S}_{\mathtt{msg}}$ be the original protocol messages that correspond to β (i.e., M_v and M'_0 share the same β). Since β binds the initiator and responder identities, we have $(u_0, v_0) = (u, v)$.

Then, we show $\mathsf{M}_0 = \mathsf{M}_u$. Let $(\alpha_0, epk_0, \sigma_0) = \mathsf{M}_0$ and $(*, \mathsf{cnt}') := h_{\mathtt{nce}}^{-1}(\alpha_0)$. Since $(u, v, \mathsf{M}_0, \mathsf{M}'_0) \in \mathcal{S}_{\mathtt{msg}}$, by the code of $\textsc{Der}_{\mathrm{R}}$, β is generated by $h_{\mathtt{nce}}(1, u, v, \mathsf{cnt}', \mathsf{cnt}_2)$ for some cnt_2, which binds the unique counter value cnt' of α_0. Conditioned on the event that CheckMsgI does not produce false positives, we have $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$. Hence, we have both $(u, \mathsf{M}_u) \in \mathcal{S}_{\mathtt{msg}}$ and $(u_0, \mathsf{M}_0) \in \mathcal{S}_{\mathtt{msg}}$. By the code in $\textsc{Init}_{\mathrm{I}}$ (cf. Figure 8), each α value has a unique counter value. α_0 and α (the one in M_u) both have the same counter value cnt', so we have $\alpha_0 = \alpha$. By a similar analysis as in the case (2) of CheckMsgI, we have $epk_0 = epk$ and $\sigma_0 = \sigma_1$, and thus $\mathsf{M}_0 = \mathsf{M}_u$.

Finally, we show that $\mathsf{M}'_0 = \mathsf{M}_v$. This is once again similar to the case (2) of CheckMsgI. By the code of $\textsc{Der}_{\mathrm{R}}$, M_v is determined by β and M_u. Since M'_0 and M_v have the same β and $\mathsf{M}_0 = \mathsf{M}_u$, we have $\mathsf{M}_v = \mathsf{M}'_0$. Therefore, we have $(u, v, \mathsf{M}_u, \mathsf{M}_v) = (u_0, v_0, \mathsf{M}_0, \mathsf{M}'_0) \in \mathcal{S}_{\mathtt{msg}}$ in this case.

By combining the above cases and the probability of false-positives from CheckMsgI, we have that if $\mathsf{CheckMsgR}(u, v, \mathsf{M}_u, \mathsf{M}_v) = 1$, then $(u, v, \mathsf{M}_u, \mathsf{M}_v) \in \mathcal{S}_{\mathtt{msg}}$ except with the probability $\frac{2N_{\mathtt{ses}}^2(|\mathcal{U}|+1)}{2^\ell}$, which concludes the proof of Lemma 7.

F.6 Proof of Lemma 8

In $\mathbf{G}_9$ (cf. Figure 9), given a query $\mathsf{H}(\mathsf{pk}_1, \mathsf{pk}_2, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k)$, we first check if the initiator and responder identities hidden in α and β match with pk_1 and pk_2, respectively. If so, let u_1 and v_2 be the initiator and responder identities recovered from α and β, respectively. Then, we use CheckMsgR to check if (α, epk, σ_1) and (β, c, σ_2) are from two matching sessions (i.e., $(u, v, \mathsf{M}_u, \mathsf{M}_v) \in \mathcal{S}_{\mathtt{msg}}$ in $\mathbf{G}_7$, cf. Figure 8). If all these checks pass, we reconstruct (epk', esk') and c' from the hidden counters, and compute $k'' := \mathsf{Decaps}(esk', c)$. We abort and return a random bit if $(epk, c, k) = (epk', c', k'')$. Thus the abort condition uses the decapsulated key, even when encapsulation and decapsulation disagree.

Before bounding the probability that **QueryH** happens, we discuss the effect of introducing CheckMsgR in H. As analyzed in $\mathbf{G}_8$, each run of CheckMsgR has error probability at most $\frac{2N_{\mathtt{ses}}(|\mathcal{U}|+1)}{2^\ell}$ (cf. Lemma 7). Since $\mathcal{A}$ can issue at most $Q_{\mathtt{H}}$ RO queries, by the union bound, the probability that $\mathcal{A}$ triggers a false-positive of CheckMsgR is at most $\frac{2Q_{\mathtt{H}}N_{\mathtt{ses}}(|\mathcal{U}|+1)}{2^\ell}$. We will add this bound in our final bound later. Here we recap that a false positive from CheckMsgR means that $\mathsf{CheckMsgR}(u, v, \mathsf{M}_u, \mathsf{M}_v) = 1$ but $(u, v, \mathsf{M}_u, \mathsf{M}_v) \notin \mathcal{S}_{\mathtt{msg}}$, namely, v never receives M_u from u or never responds with M_v to (u, M_u).

Then, to bound the probability that **QueryH** happens in $\mathbf{G}_9$, we construct a reduction $\mathcal{B}_{\mathsf{KEM}}$ in Figure 19 which is against the OW-PCA security of KEM. In the construction of $\mathcal{B}_{\mathsf{KEM}}$, we use the oracle access to $\textsc{ChalPk}$ and $\textsc{ChalCt}$ to generate all epk and c in $\textsc{Init}_{\mathrm{I}}$ and $\textsc{Der}_{\mathrm{R}}$, respectively. Now the counter cnt indexes the challenge epk and c from $\textsc{ChalPk}$ and $\textsc{ChalCt}$, respectively. Moreover, we also use the counter value hidden in α and β to recover epk and c from $\textsc{ChalPk}$ and $\textsc{ChalCt}$, respectively, in CheckMsgI, CheckMsgR, and H. This allows us to use the $\textsc{Pco}$ oracle to check if any RO query contains a solution of the OW-PCA game. Note that we do not need to maintain $\mathcal{S}_{\mathtt{sid}}$, $\mathcal{S}_{\mathtt{test}}$, and $\mathcal{S}_{\mathtt{cor}}$ and simulate Fresh and Valid since it is unnecessary to continue the game if all $\mathcal{A}$'s queries cannot be used to break the OW-PCA security of KEM. This key observation allows the reduction to forget these lists and achieve a memory-tight simulation.

To simulate the abort test, $\mathcal{B}_{\mathsf{KEM}}$ obtains epk' from $\textsc{ChalPk}(\mathsf{cnt}_1)$ and c' from $\textsc{ChalCt}(\mathsf{cnt}_1, \mathsf{cnt}_2)$. These oracles return the public key and ciphertext, not the secret key. After checking $(epk, c) = (epk', c')$, the reduction queries $\textsc{Pco}(\mathsf{cnt}_1, c, k)$ using the key supplied in $\mathcal{A}$'s hash query. An answer of 1 certifies precisely the decapsulation-key equality used by the game. The reduction then outputs $(\mathsf{cnt}_1, \mathsf{cnt}_2, k)$: the second component is the challenge ciphertext index. Subject to the recognition error just discussed, every **QueryH** event therefore gives a winning output in the OW-PCA experiment. Hence we have

$$\begin{aligned}
&\left|\Pr\left[\mathbf{G}_8^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_9^{\mathcal{A}} \Rightarrow 1\right]\right| \leq \Pr\left[\textbf{QueryH} \text{ in } \mathbf{G}_9\right] \\
&\leq \Pr[\mathsf{OW\text{-}PCA}_{\mathsf{KEM}}^{\mathcal{B}_{\mathsf{KEM}}} \Rightarrow 1] + \frac{2Q_{\mathtt{H}}N_{\mathtt{ses}}(|\mathcal{U}|+1)}{2^\ell}
\end{aligned}$$

Reduction $\mathcal{B}_{\mathsf{KEM}}^{\textsc{ChalPk},\textsc{ChalCt},\textsc{Pco}}(\mathsf{par}_{\mathtt{kem}})$

00 $\mathsf{cnt} := 0$, $b \leftarrow \{0,1\}$
01 Initialize $f_{\mathtt{seq}}, f_{\mathtt{H}}, f_{\mathtt{skg}}, f_{\mathtt{sI}}, f_{\mathtt{sR}}, f_{\mathtt{ekg}}$, $f_{\mathtt{enc}}, f_{\mathtt{rSK}}, h_{\mathtt{nce}}$, and $h_{\mathtt{seq}}$ as in $\mathbf{G}_3$
02 $f_{\mathtt{k}} \leftarrow \mathsf{Func}(\{0,1\}^*, \mathcal{K})$
03 $O_1 := (\textsc{InitI}, \textsc{DerI}, \textsc{DerR})$
04 $O_2 := (\textsc{Pk}, \textsc{Cor}, \textsc{Rev}, \textsc{Test})$
05 $\mathsf{par}_{\mathtt{sig}} \leftarrow \mathsf{SIG.Setup}$, $\mathsf{par} := (\mathsf{par}_{\mathtt{kem}}, \mathsf{par}_{\mathtt{sig}})$
06 $b' \leftarrow \mathcal{A}^{O_1,O_2,\mathsf{H}}(\mathsf{par})$
07 **return** $\bot$ // **QueryH** never happens

Oracle $\textsc{InitI}(u, v)$

08 $\mathsf{cnt} := \mathsf{cnt} + 1$, $\mathsf{seq}_1 := h_{\mathtt{seq}}(u, v, \mathsf{cnt})$
09 $\alpha := h_{\mathtt{nce}}(0, u, v, 0, \mathsf{cnt})$
10 $epk \leftarrow \textsc{ChalPk}(\mathsf{cnt})$
11 $\sigma := \mathsf{Sign}(\mathsf{sk}_u, (u,\alpha, epk); f_{\mathtt{sI}}(u,\alpha, epk))$
12 $\mathsf{M}_u := (\alpha, epk, \sigma)$
13 **return** $(\mathsf{seq}_1, \mathsf{M}_u)$

Oracle $\textsc{DerI}(u, v, \mathsf{seq}_1, \mathsf{M}_v = (\beta, c, \sigma_2))$

14 $(u', v', \mathsf{cnt}') := h_{\mathtt{seq}}^{-1}(\mathsf{seq}_1)$
15 **if** $(u', v') \neq (u, v) \vee \mathsf{cnt} < \mathsf{cnt}'$: **return** $\bot$
16 $\alpha := h_{\mathtt{nce}}(0, u, v, 0, \mathsf{cnt}')$
17 $epk \leftarrow \textsc{ChalPk}(\mathsf{cnt}')$
18 $\sigma_1 := \mathsf{Sign}(\mathsf{sk}_u, (u,\alpha, epk); f_{\mathtt{sI}}(u,\alpha, epk))$
19 **if** $\mathsf{Vrfy}(\mathsf{pk}_v, (u, v,\alpha, \beta, epk, c, \sigma_1), \sigma_2) = 0$:
20 **return** $\bot$
21 $\mathsf{M}_u := (\alpha, epk, \sigma_1)$
22 $\mathsf{seq}_2 := f_{\mathtt{seq}}(\mathsf{seq}_1, u, v, \mathsf{M}_u, \mathsf{M}_v)$
23 $\mathsf{sID} := (\mathsf{seq}_1, \mathsf{seq}_2, u, v, \mathsf{M}_u, \mathsf{M}_v)$
24 **return** sID

Oracle $\textsc{DerR}(u, v, \mathsf{M}_u = (\alpha, epk, \sigma_1))$

25 **if** $\mathsf{Vrfy}(\mathsf{pk}_u, (u,\alpha, epk), \sigma_1) = 0$: **return** $\bot$
26 $\mathsf{cnt} := \mathsf{cnt} + 1$
27 **if** $\mathsf{CheckMsgI}(u, \mathsf{M}_u)$
28 $(*, \mathsf{cnt}_1) := h_{\mathtt{nce}}^{-1}(\alpha)$
29 $\beta := h_{\mathtt{nce}}(1, u, v, \mathsf{cnt}_1, \mathsf{cnt})$
30 $c \leftarrow \textsc{ChalCt}(\mathsf{cnt}_1, \mathsf{cnt})$
31 **else**
32 $\beta := h_{\mathtt{nce}}(1, u, v, 0, \mathsf{cnt})$
33 $(c, k) := \mathsf{KEM.Encaps}(epk; f_{\mathtt{enc}}(epk, \mathsf{cnt}))$
34 $r_{\mathtt{sign}} := f_{\mathtt{sR}}(u, v,\alpha, \beta, epk, c, \sigma_1)$
35 $\sigma_2 := \mathsf{Sign}(\mathsf{sk}_v, (u, v,\alpha, \beta, epk, c, \sigma_1); r_{\mathtt{sign}})$
36 $\mathsf{M}_v := (\beta, c, \sigma_2)$
37 $\mathsf{seq} := f_{\mathtt{seq}}(u, v, \mathsf{M}_u, \mathsf{M}_v)$
38 $\mathsf{sID} := (\mathsf{seq}, u, v, \mathsf{M}_u, \mathsf{M}_v)$
39 **return** sID

Alg $\mathsf{CheckMsgI}(u, \mathsf{M}_u = (\alpha, epk, \sigma_1))$

40 $(b', u', v, \mathsf{cnt}', \mathsf{cnt}) := h_{\mathtt{nce}}^{-1}(\alpha)$
41 **if** $(b', u', v, \mathsf{cnt}', \mathsf{cnt}) \neq \bot \wedge u = u'$
$\wedge\ \mathsf{cnt} \in [N_{\mathtt{ses}}] \wedge b' = 0 \wedge \mathsf{cnt}' = 0$
42 $epk' \leftarrow \textsc{ChalPk}(\mathsf{cnt})$
43 $\sigma_1' := \mathsf{Sign}(\mathsf{sk}_u, (u,\alpha, epk); f_{\mathtt{sI}}(u,\alpha, epk))$
44 **if** $epk' = epk \wedge \sigma_1' = \sigma_1$: **return** 1
45 **return** 0

Alg $\mathsf{CheckMsgR}(u, v, \mathsf{M}_u, \mathsf{M}_v)$

46 $(\alpha, epk, \sigma_1) := \mathsf{M}_u$, $(\beta, c, \sigma_2) := \mathsf{M}_v$
47 **if** $\neg\mathsf{CheckMsgI}(u, \mathsf{M}_u)$: **return** 0
48 $(b', u', v', \mathsf{cnt}_1', \mathsf{cnt}_2) := h_{\mathtt{nce}}^{-1}(\beta)$
49 $(*, \mathsf{cnt}_1) := h_{\mathtt{nce}}^{-1}(\alpha)$
50 **if** $(b', u', v', \mathsf{cnt}_1', \mathsf{cnt}_2) \neq \bot$
$\wedge\ b' = 1 \wedge (u, v) = (u', v')$
$\wedge\ \mathsf{cnt}_2 \in [N_{\mathtt{ses}}] \wedge \mathsf{cnt}_1' = \mathsf{cnt}_1$
51 $c' \leftarrow \textsc{ChalCt}(\mathsf{cnt}_1, \mathsf{cnt}_2)$
52 **if** $c' \neq c$: **return** 0
53 $r_{\mathtt{sign}} := f_{\mathtt{sR}}(u, v,\alpha, \beta, epk, c, \sigma_1)$
54 $\sigma_2' := \mathsf{Sign}(\mathsf{sk}_v, (u, v,\alpha, \beta, epk, c, \sigma_1); r_{\mathtt{sign}})$
55 **if** $\sigma_2' = \sigma_2$: **return** 1
56 **return** 0

Oracle $\mathsf{H}(\mathsf{pk}_1, \mathsf{pk}_2, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k)$

57 $k^* := k$, $d := 0$
58 $(b_1, u_1, v_1, \mathsf{cnt}_1', \mathsf{cnt}_1) := h_{\mathtt{nce}}^{-1}(\alpha)$
59 $(b_2, u_2, v_2, \mathsf{cnt}_2', \mathsf{cnt}_2) := h_{\mathtt{nce}}^{-1}(\beta)$
60 $\mathsf{M}_u := (\alpha, epk, \sigma_1)$, $\mathsf{M}_v := (\beta, c, \sigma_2)$
61 **if** $h_{\mathtt{nce}}^{-1}(\alpha) \neq \bot \wedge h_{\mathtt{nce}}^{-1}(\beta) \neq \bot$:
62 $(\mathsf{pk}_1', *) := \mathsf{GetKey}(u_1)$
63 $(\mathsf{pk}_2', *) := \mathsf{GetKey}(v_2)$
64 **if** $\mathsf{pk}_1 = \mathsf{pk}_1' \wedge \mathsf{pk}_2 = \mathsf{pk}_2'$
$\wedge\ \mathsf{CheckMsgR}(u_1, v_2, \mathsf{M}_u, \mathsf{M}_v)$:
65 $epk' \leftarrow \textsc{ChalPk}(\mathsf{cnt}_1)$
66 $c' \leftarrow \textsc{ChalCt}(\mathsf{cnt}_1, \mathsf{cnt}_2)$
67 **if** $(epk, c) = (epk', c') \wedge\ \textsc{Pco}(\mathsf{cnt}_1, c, k)$:
68 $\mathtt{Flag}_{\mathtt{queryH}} := \mathbf{true}$
69 **abort** and **output** $(\mathsf{cnt}_1, \mathsf{cnt}_2, k)$
70 **if** $h_{\mathtt{nce}}^{-1}(\alpha) \neq \bot\ \wedge\ b_1 = 0$:
71 $epk' \leftarrow \textsc{ChalPk}(\mathsf{cnt}_1)$
72 **if** $epk = epk'$:
73 $d := 1$
74 **if** $\textsc{Pco}(\mathsf{cnt}_1, c, k)$: $k^* := f_{\mathtt{k}}(epk, c)$
75 **if** $d = 0\ \wedge\ h_{\mathtt{nce}}^{-1}(\beta) \neq \bot\ \wedge\ b_2 = 1$:
76 $y := f_{\mathtt{enc}}(epk, \mathsf{cnt}_2)$
77 $(c', k') := \mathsf{KEM.Encaps}(epk; y)$
78 **if** $c = c'\ \wedge\ k = k'$: $k^* := f_{\mathtt{k}}(epk, c)$
79 **return** $f_{\mathtt{H}}(\mathsf{pk}_1, \mathsf{pk}_2, \alpha, \beta, epk, c, \sigma_1, \sigma_2, k^*)$

Fig. 19. Reduction $\mathcal{B}_{\mathsf{KEM}}$ against the OW-PCA game for $\mathbf{G}_9$ in Theorem 1. $\mathcal{B}_{\mathsf{KEM}}$ has access to ($\textsc{ChalPk}$, $\textsc{ChalCt}$, $\textsc{Pco}$). Highlighted lines show how we use these oracles to simulate the AKE game for $\mathcal{A}$. $\textsc{Pk}$ and $\textsc{Cor}$ use GetKey as in Figure 4. The oracles $\textsc{Rev}$ and $\textsc{Test}$, with CheckSid and ComputeSK, are those of $\mathbf{G}_6$ in Figure 7. We omit all updates to $\mathcal{S}_{\mathtt{sid}}$, $\mathcal{S}_{\mathtt{test}}$, $\mathcal{S}_{\mathtt{rev}}$, and $\mathcal{S}_{\mathtt{cor}}$: these sets are read only by the final checks, which this reduction does not run.

Note that we do not need to count in the decryption error probability of KEM since we use $\textsc{Pco}(\mathsf{cnt}, c, k)$ to replace $k = \mathsf{Decaps}(esk, c)$, where (epk, esk) is the challenge key pair with index cnt from the OW-PCA challenges. By the definition of the OW-PCA game (cf. Figure 1), $\textsc{Pco}$ uses esk to decapsulate c and checks if the output equals k. Hence, using $\textsc{Pco}$ does not cause any additional decryption error probability.

Finally, we analyze the reduced running time and memory usage of $\mathcal{B}_{\mathsf{KEM}}$. Let $\mathbf{T}(\mathtt{aSigKEM})$ and $\mathbf{Mem}(\mathtt{aSigKEM})$ be the required running time and memory for running a complete protocol execution of

$\mathtt{aSigKEM}$, respectively. By the construction of $\mathcal{B}_{\mathsf{KEM}}$, we have

$$\begin{aligned}\mathbf{Mem}^*(\mathcal{B}_{\mathsf{KEM}}) = \mathbf{Mem}(\mathcal{A}) &+ \mathrm{O}(\log N_{\mathtt{ses}}) + \mathrm{O}(\ell) \\ &+ \mathrm{O}(\log|\mathcal{U}|) + \mathrm{O}(\log|\mathcal{SEQ}|) + \mathbf{Mem}(\mathtt{aSigKEM})\end{aligned}$$

Each of the $Q_{\mathtt{query}}$ non-RO queries and $Q_{\mathtt{H}}$ queries to H triggers only a constant number of primitive operations and auxiliary calls. This includes repeated long-term key generation for Pk and Cor, and public-key recovery in $\mathsf{ComputeSK}$ for Rev and Test. Internal calls are charged to the external query that triggers them, using the cost convention of Theorem 1. Hence, we have

$$\mathbf{T}^*(\mathcal{B}_{\mathsf{KEM}}) = \mathbf{T}(\mathcal{A}) + \mathrm{O}(Q_{\mathtt{query}} + Q_{\mathtt{H}}) \cdot \mathbf{T}(\mathtt{aSigKEM})$$

F.7 Proof of Lemma 9

In $\mathbf{G}_{10}$, if the event **Forge** does not happen, then $\mathcal{A}$ cannot distinguish $\mathbf{G}_9$ from $\mathbf{G}_{10}$ since the distributions of $\textsc{Der}_{\mathrm{I}}$ and $\textsc{Der}_{\mathrm{R}}$ do not change unless $\mathcal{A}$ triggers **Forge**. Therefore, we only need to bound the probability that **Forge** happens in $\mathbf{G}_{10}$. Specifically, **Forge** happens if $\mathtt{Flag}_{\mathtt{forge}}$ is raised as **true** at any point of the game, which means that at least one of the following cases happens:

- $\mathcal{A}$ queries $\textsc{Der}_{\mathrm{R}}(u, v, \mathsf{M}_u)$, where $\mathsf{M}_u = (\alpha, epk, \sigma_1)$, u is uncorrupted, $\mathsf{CheckMsgI}(u, \mathsf{M}_u) = 0$, and $\mathsf{Vrfy}(\mathsf{pk}_u, (u, \alpha, epk), \sigma_1) = 1$; or
- $\mathcal{A}$ queries $\textsc{Der}_{\mathrm{I}}(u, v, \mathsf{seq}_1, \mathsf{M}_v)$, where $\mathsf{M}_v = (\beta, c, \sigma_2)$, the sequence-number check passes, v is uncorrupted, $\mathsf{CheckMsgR}(u, v, \mathsf{M}_u, \mathsf{M}_v) = 0$, and $\mathsf{Vrfy}(\mathsf{pk}_v, (u, v, \alpha, \beta, epk, c, \sigma_1), \sigma_2) = 1$. Here $\mathsf{M}_u = (\alpha, epk, \sigma_1)$ is reconstructed by $\textsc{Der}_{\mathrm{I}}$ from (u, v, seq_1).

To bound $\Pr[\mathbf{Forge}]$, we construct a reduction $\mathcal{B}_{\mathsf{SIG}}$ in Figure 20 which is against the MU^c-sEUF-1CMA security of SIG. Since $\mathbf{G}_9$, there is an abort event defined in H. If this event happens, then $\mathcal{B}_{\mathsf{SIG}}$ simply aborts the simulation. Note that the occurrence of **Forge** means that **QueryH** does not happen, since otherwise, the game would have already aborted at H. Hence, in the later analysis, we can assume that **QueryH** does not happen in $\mathbf{G}_{10}$.

We first explain how $\mathcal{B}_{\mathsf{SIG}}$ simulates $\mathbf{G}_{10}$ for $\mathcal{A}$ *before* **Forge** *happens*. In the MU^c-sEUF-1CMA security game, $\mathcal{B}_{\mathsf{SIG}}$ does not have access to the secret keys of uncorrupted users. To simulate signatures, $\mathcal{B}_{\mathsf{SIG}}$ uses the signing oracle $\textsc{Sign}'$ to generate signatures for all users. By the definition of MU^c-sEUF-1CMA, repeated queries to $\textsc{Sign}'$ return the same signature. Hence, all derandomized signatures in $\mathbf{G}_{10}$ have the same distribution as those generated by $\textsc{Sign}'$. The user public key oracle and corruption oracle are also simulated by $\textsc{Pk}'$ and $\textsc{Cor}'$, respectively.

An important difference between the construction of $\mathcal{B}_{\mathsf{SIG}}$ and $\mathbf{G}_{10}$ is on how to handle the event **Forge**. In $\mathbf{G}_{10}$, the game explicitly determines whether **Forge** happens by recording corrupted users and using the auxiliary algorithms $\mathsf{CheckMsgI}$ and $\mathsf{CheckMsgR}$. In the construction of $\mathcal{B}_{\mathsf{SIG}}$, we do not need to explicitly determine whether **Forge** happens or not, thanks to the definition of MU^c-sEUF-1CMA security. In the MU^c-sEUF-1CMA game, the $\textsc{Forge}'$ oracle checks whether the forgery produced by $\mathcal{A}$ is valid and fresh, which includes checking whether the user is uncorrupted (cf. line 12). This oracle can be queried multiple times, and $\mathcal{B}_{\mathsf{SIG}}$ wins the MU^c-sEUF-1CMA game as long as at least one of these queries is a valid and fresh forgery. This observation allows $\mathcal{B}_{\mathsf{SIG}}$ to "forget" the corrupted user set $\mathcal{S}_{\mathtt{cor}}$. It remains to show freshness with respect to all $\textsc{Sign}'$ queries, including those made inside the checking procedures. Every signature on (u, α, epk) obtained under u by $\textsc{Init}_{\mathrm{I}}$, $\textsc{Der}_{\mathrm{I}}$, or $\mathsf{CheckMsgI}$ satisfies the structural checks of $\mathsf{CheckMsgI}$. Since repeated signing queries return the same signature, replaying such a signature makes $\mathsf{CheckMsgI}$ return 1. Thus a valid signature rejected by this procedure is fresh.

For $\textsc{Der}_{\mathrm{I}}$, the reconstructed M_u satisfies $\mathsf{CheckMsgI}(u, \mathsf{M}_u) = 1$. Suppose its candidate pair (m_2, σ_2), with $m_2 = (u, v, \alpha, \beta, epk, c, \sigma_1)$, was previously returned by $\textsc{Sign}'(v, m_2)$. If this query came from $\mathsf{CheckMsgR}$, the structural checks already passed. If it came from $\textsc{Der}_{\mathrm{R}}$, the identities and the complete initiator message in m_2 force that invocation to use the same (u, v, M_u). It therefore used the recognized-message branch and generated the tag and ciphertext expected by $\mathsf{CheckMsgR}$. In either case, $\mathsf{CheckMsgR}$ returns 1, a contradiction. A different valid signature on a previously signed message is also fresh under strong unforgeability. Hence, at the first **Forge**, the submitted signature is valid and fresh for an uncorrupted user. This argument requires no signing-history table in the reduction.

Reduction $\mathcal{B}_{\mathsf{SIG}}^{\textsc{Pk}',\textsc{Sign}',\textsc{Cor}',\textsc{Forge}'}(\mathsf{par}_{\mathtt{sig}})$

00 $\mathsf{cnt} := 0$, $b \leftarrow \{0,1\}$
01 Initialize $f_{\mathtt{seq}}, f_{\mathtt{H}}, f_{\mathtt{skg}}, f_{\mathtt{sI}}, f_{\mathtt{sR}}, f_{\mathtt{ekg}}$, $f_{\mathtt{enc}}, f_{\mathtt{rSK}}, f_{\mathtt{k}}, h_{\mathtt{nce}}$, and $h_{\mathtt{seq}}$ as in $\mathbf{G}_6$
02 $\mathsf{par}_{\mathtt{kem}} \leftarrow \mathsf{KEM.Setup}$
03 $\mathsf{par} := (\mathsf{par}_{\mathtt{kem}}, \mathsf{par}_{\mathtt{sig}})$
04 $b' \leftarrow \mathcal{A}^{O,\mathsf{H}}(\mathsf{par})$

Alg $\mathsf{CheckMsgI}(u, \mathsf{M}_u = (\alpha, epk, \sigma_1))$

05 $(b', u', v, \mathsf{cnt}', \mathsf{cnt}) := h_{\mathtt{nce}}^{-1}(\alpha)$
06 **if** $(b', u', v, \mathsf{cnt}', \mathsf{cnt}) \neq \bot \wedge u = u'$
$\wedge\ \mathsf{cnt} \in [N_{\mathtt{ses}}] \wedge b' = 0 \wedge \mathsf{cnt}' = 0$
07 $(epk', esk') := \mathsf{KEM.KG}(; f_{\mathtt{ekg}}(\mathsf{cnt}))$
08 **if** $epk' \neq epk$: **return** 0
09 $\sigma_1' := \textsc{Sign}'(u, (u,\alpha, epk))$
10 **if** $\sigma_1' = \sigma_1$: **return** 1
11 **return** 0

Alg $\mathsf{CheckMsgR}(u, v, \mathsf{M}_u, \mathsf{M}_v)$

12 $(\alpha, epk, \sigma_1) := \mathsf{M}_u$, $(\beta, c, \sigma_2) := \mathsf{M}_v$
13 **if** $\neg\mathsf{CheckMsgI}(u, \mathsf{M}_u)$: **return** 0
14 $(b', u', v', \mathsf{cnt}_1', \mathsf{cnt}_2) := h_{\mathtt{nce}}^{-1}(\beta)$
15 $(*, \mathsf{cnt}_1) := h_{\mathtt{nce}}^{-1}(\alpha)$
16 **if** $(b', u', v', \mathsf{cnt}_1', \mathsf{cnt}_2) \neq \bot$
$\wedge\ b' = 1 \wedge (u, v) = (u', v')$
$\wedge\ \mathsf{cnt}_2 \in [N_{\mathtt{ses}}] \wedge \mathsf{cnt}_1' = \mathsf{cnt}_1$
17 $(epk', esk') := \mathsf{KEM.KG}(; f_{\mathtt{ekg}}(\mathsf{cnt}_1))$
18 $y := f_{\mathtt{enc}}(epk', \mathsf{cnt}_2)$
19 $(c', k') := \mathsf{KEM.Encaps}(epk'; y)$
20 **if** $c' \neq c$: **return** 0
21 $\sigma_2' := \textsc{Sign}'(v, (u, v,\alpha, \beta, epk, c', \sigma_1))$
22 **if** $\sigma_2' = \sigma_2$: **return** 1
23 **return** 0

Oracle $\textsc{Pk}(u)$

24 **return** $\textsc{Pk}'(u)$

Oracle $\textsc{Cor}(u)$

25 $(*, \mathsf{sk}_u) := \textsc{Cor}'(u)$
26 **return** sk_u

Oracle $\textsc{Init}_{\mathrm{I}}(u, v)$

27 $\mathsf{cnt} := \mathsf{cnt} + 1$, $\mathsf{seq}_1 := h_{\mathtt{seq}}(u, v, \mathsf{cnt})$
28 $\alpha := h_{\mathtt{nce}}(0, u, v, 0, \mathsf{cnt})$
29 $(epk, esk) := \mathsf{KEM.KG}(; f_{\mathtt{ekg}}(\mathsf{cnt}))$
30 $\sigma := \textsc{Sign}'(u, (u,\alpha, epk))$
31 $\mathsf{M}_u := (\alpha, epk, \sigma)$
32 **return** $(\mathsf{seq}_1, \mathsf{M}_u)$

Oracle $\textsc{Der}_{\mathrm{R}}(u, v, \mathsf{M}_u = (\alpha, epk, \sigma_1))$

33 **if** $\mathsf{Vrfy}(\mathsf{pk}_u, (u,\alpha, epk), \sigma_1) = 0$: **return** $\bot$
34 **if** $\neg\mathsf{CheckMsgI}(u, \mathsf{M}_u)$
35 $\textsc{Forge}'(u, (u,\alpha, epk), \sigma_1)$
36 $\mathsf{cnt} := \mathsf{cnt} + 1$
37 **if** $\mathsf{CheckMsgI}(u, \mathsf{M}_u)$
38 $(*, \mathsf{cnt}_1) := h_{\mathtt{nce}}^{-1}(\alpha)$
39 $\beta := h_{\mathtt{nce}}(1, u, v, \mathsf{cnt}_1, \mathsf{cnt})$
40 **else** $\beta := h_{\mathtt{nce}}(1, u, v, 0, \mathsf{cnt})$
41 $(c, k) := \mathsf{KEM.Encaps}(epk; f_{\mathtt{enc}}(epk, \mathsf{cnt}))$
42 $\sigma_2 := \textsc{Sign}'(v, (u, v,\alpha, \beta, epk, c, \sigma_1))$
43 $\mathsf{M}_v := (\beta, c, \sigma_2)$, $\mathsf{seq} = f_{\mathtt{seq}}(u, v, \mathsf{M}_u, \mathsf{M}_v)$
44 $\mathsf{sID} := (\mathsf{seq}, u, v, \mathsf{M}_u, \mathsf{M}_v)$
45 **return** sID

Oracle $\textsc{Der}_{\mathrm{I}}(u, v, \mathsf{seq}_1, \mathsf{M}_v = (\beta, c, \sigma_2))$

46 $(u', v', \mathsf{cnt}') := h_{\mathtt{seq}}^{-1}(\mathsf{seq}_1)$
47 **if** $(u', v') \neq (u, v) \vee \mathsf{cnt} < \mathsf{cnt}'$: **return** $\bot$
48 $\alpha := h_{\mathtt{nce}}(0, u, v, 0, \mathsf{cnt}')$
49 $(epk, esk) := \mathsf{KEM.KG}(; f_{\mathtt{ekg}}(\mathsf{cnt}'))$
50 $\sigma_1 := \textsc{Sign}'(u, (u,\alpha, epk))$
51 $\mathsf{M}_u := (\alpha, epk, \sigma_1)$
52 **if** $\mathsf{Vrfy}(\mathsf{pk}_v, (u, v,\alpha, \beta, epk, c, \sigma_1), \sigma_2) = 0$: **return** $\bot$
53 **if** $\neg\mathsf{CheckMsgR}(u, v, \mathsf{M}_u, \mathsf{M}_v)$
54 $\textsc{Forge}'(v, (u, v,\alpha, \beta, epk, c, \sigma_1), \sigma_2)$
55 $\mathsf{seq}_2 := f_{\mathtt{seq}}(\mathsf{seq}_1, u, v, \mathsf{M}_u, \mathsf{M}_v)$
56 $\mathsf{sID} := (\mathsf{seq}_1, \mathsf{seq}_2, u, v, \mathsf{M}_u, \mathsf{M}_v)$
57 **return** sID

Fig. 20. $\mathcal{B}_{\mathsf{SIG}}$ against MU^c-$\mathsf{sEUF\text{-}1CMA}$ for $\mathbf{G}_{10}$ in Theorem 1. $\mathcal{B}_{\mathsf{SIG}}$ has access to $\{\textsc{Pk}', \textsc{Cor}', \textsc{Sign}', \textsc{Forge}'\}$. $\mathcal{A}$ has access to $O := \{\textsc{Init}_{\mathrm{I}}, \textsc{Der}_{\mathrm{I}}, \textsc{Der}_{\mathrm{R}}, \textsc{Pk}, \textsc{Cor}, \textsc{Rev}, \textsc{Test}\}$. Highlighted lines show how we use these oracles to simulate $\mathbf{G}_{10}$ for $\mathcal{A}$. Every required public key is obtained from $\textsc{Pk}'$. The oracles $\textsc{Rev}$ and $\textsc{Test}$, with $\mathsf{CheckSid}$ and $\mathsf{ComputeSK}$, are those of $\mathbf{G}_6$ in Figure 7, with their history-set updates omitted. We retain no session, test, reveal, or corruption set, since the reduction does not run the final freshness or validity checks. H is the same as in $\mathbf{G}_9$ (cf. Figure 9), using the public-key oracle and message-checking procedures of this reduction.

Therefore, $\mathcal{B}_{\mathsf{SIG}}$ correctly simulates the game $\mathbf{G}_{10}$ in $\mathcal{A}$'s view until **Forge** happens and wins whenever this event occurs, so we have

$$\left|\Pr\left[\mathbf{G}_9^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathbf{G}_{10}^{\mathcal{A}} \Rightarrow 1\right]\right| \leq \Pr\left[\mathbf{Forge}\right] \leq \Pr[\mathsf{MU}^c\text{-}\mathsf{sEUF\text{-}1CMA}_{\mathsf{SIG}}^{\mathcal{B}_{\mathsf{SIG}}} \Rightarrow 1]$$

We also remark that, after **Forge** happens, $\mathcal{B}_{\mathsf{SIG}}$ continues to simulate the game for $\mathcal{A}$ until $\mathcal{A}$ outputs its bit b'. Although the simulation after **Forge** happens is no longer perfect, it does not change the fact that $\mathcal{B}_{\mathsf{SIG}}$ has found a valid and fresh forgery. As long as the simulation of $\mathbf{G}_{10}$ is correct before **Forge** happens, the winning advantage of $\mathcal{B}_{\mathsf{SIG}}$ in bounding $\mathbf{G}_9$ and $\mathbf{G}_{10}$ is valid. Similar to $\mathcal{B}_{\mathsf{KEM}}$, $\mathcal{B}_{\mathsf{SIG}}$ does not need to maintain $\mathcal{S}_{\mathtt{cor}}$ and $\mathcal{S}_{\mathtt{rev}}$ since it is necessary to continue the game if $\mathcal{A}$ never forges any valid and fresh signatures.

The identities used in signatures are available from the current query or recovered by $h_{\mathtt{nce}}^{-1}$ and $h_{\mathtt{seq}}^{-1}$. Thus their inclusion requires no additional user or session table and preserves the reduced-memory bounds of both reductions.

Analysis of Time and Memory. Finally, we analyse the (reduced) running time and memory usage of $\mathcal{B}_{\mathsf{SIG}}$. Let $\mathbf{T}(\mathtt{aSigKEM})$ and $\mathbf{Mem}(\mathtt{aSigKEM})$ be the required running time and memory for running a complete protocol execution of $\mathtt{aSigKEM}$, respectively. By the constructions of $\mathcal{B}_{\mathsf{SIG}}$ and a similar analysis for $\mathcal{B}_{\mathsf{KEM}}$ in Supp. Mat. F.6, we have

$$\begin{aligned}\mathbf{Mem}^*(\mathcal{B}_{\mathsf{SIG}}) &= \mathbf{Mem}(\mathcal{A}) + \mathrm{O}(\log N_{\mathtt{ses}}) + \mathrm{O}(\ell) \\ &\quad + \mathrm{O}(\log |\mathcal{U}|) + \mathrm{O}(\log |\mathcal{SEQ}|) + \mathbf{Mem}(\mathtt{aSigKEM}) \\ \mathbf{T}^*(\mathcal{B}_{\mathsf{SIG}}) &= \mathbf{T}(\mathcal{A}) + \mathrm{O}(Q_{\mathtt{query}} + Q_{\mathtt{H}}) \cdot \mathbf{T}(\mathtt{aSigKEM})\end{aligned}$$

G Use of AI Tools

The authors used ChatGPT (GPT-5.6 Sol) and OpenAI Codex (GPT-6 Astra) to assist with revising this manuscript. ChatGPT reviewed the manuscript, suggested corrections to language, notation, references, and pseudocode, and checked consistency between theorem statements and proofs. The authors implemented revisions based on these suggestions either themselves or with assistance from Codex. Codex also helped implement revisions based on the reviewers' comments in the LaTeX source.

The assistance included correcting spelling and grammar, query counts, and statistical terms in several security bounds. The tools also helped check and correct inconsistencies in mathematical notation, algorithm inputs, variable usage, and references to security games. The authors reviewed and double-checked the changes made with these tools and take responsibility for the final manuscript.