Updatable Public-Key Encryption, Revisited

Joël Alwen[1], Georg Fuchsbauer[2], and Marta Mularczyk[1]

[1] AWS Wickr, `{alwenjo,mulmarta}@amazon.com`
[2] TU Wien, `first.last@tuwien.ac.at`

Abstract. We revisit Updatable Public-Key Encryption (UPKE), which was introduced as a practical mechanism for building forward-secure cryptographic protocols. We begin by observing that all UPKE notions to date are neither syntactically flexible nor secure enough for the most important multi-party protocols motivating UPKE. We provide an intuitive taxonomy of UPKE properties – some partially or completely overlooked in the past – along with an overview of known (explicit and implicit) UPKE constructions. We then introduce a formal UPKE definition capturing all intuitive properties needed for multi-party protocols. Next, we provide a practical pairing-based construction for which we provide concrete security bounds under a standard assumption in the random oracle and the algebraic group model. The efficiency profile of the scheme compares very favorably with existing UPKE constructions (despite the added flexibility and stronger security). For example, when used to improve the forward security of the Messaging Layer Security protocol [RFC9420], our new UPKE construction requires $\approx 1\%$ of the bandwidth of the next-most efficient UPKE construction satisfying the strongest UPKE notion previously considered.

Table of Contents

1 Introduction

Spurred on by the seemingly never-ending procession of data breaches, 0-day exploits and system compromises, it is becoming ever more important in applied cryptography to design protocols with the ability to automatically limit the blast radii of key and state compromises. Among other techniques, this has lead to interest in primitives designed to provide cheap but effective *forward security*, namely the property that security holds despite possible future compromises.

A naïve (though not ineffective) approach to providing forward security for, say, public-key encryption (PKE) is for the owner of a key pair (pk, sk) to periodically sample a fresh and independent key pair (pk', sk') that replaces its old keys. While this technique does provide forward security – old ciphertexts encrypted to pk remain secure even if the adversary learns sk' – it comes with a serious drawback from the protocol perspective. After each key rotation the receiver must first inform prospective senders of the new public key before new messages can be sent privately to the receiver again.[3] Besides increasing communication complexity, the biggest issue with this is that it forces potentially onerous coordination requirements on protocol participants.

Avoiding this cost motivated the study of *Puncturable Public-Key Encryption* [GM15] (PPKE) as a stand-alone primitive. PPKE provides essentially the same security as the naïve approach but without further coordination between parties beyond the initial public key distribution. After that, any number of senders may independently send any number of ciphertexts to the receiver which can be delivered in any order (or not at all). Despite the lack of coordination between parties, PPKE guarantees that at any point, leaking the receiver's secret key reveals nothing about messages in ciphertexts it had already received and decrypted.

Clearly a powerful tool for building forward-secure protocols, PPKE lies at the heart of recent forward-secure 0-round trip key agreement protocols [GHJL17]. But minimizing round and communication complexity for forward-secure key agreement underpins other classes of cryptographic protocols. Notably, these include 2-party ratcheting [JS18, PR18, JMM19, DV19, CCD$^+$20], the multi-party analogue: continuous group key agreement (CGKA) [ACDT20, AAN$^+$22, ACJM20] and secure group and 2-party messaging [ACDT21]. In this work, we are especially interested in CGKA and secure group messaging (SGM) applications of forward-secure encryption primitives as these demand new, and hitherto seemingly overlooked, properties of the underlying primitive.

Updatable public key encryption. Unfortunately, despite its wide-ranging practical applications, to date, PPKE constructions are not practically efficient for many real-world use cases, in particular in the ratcheting and messaging settings. This has given rise to a new class of "off-brand" forward-secure encryption schemes in the messaging literature called *Updatable Public-Key Encryption* (UPKE). They aim for a happy middle ground between forward secrecy with minimal interaction and truly practical efficiency.

Intuitively, UPKE is public-key encryption where senders can also generate *update tokens*. Applying an update token up to a public key pk produces an updated public key $pk \rightarrow_{up} pk'$. Similarly, applying up to the secret key sk of pk yields the secret key $sk \rightarrow_{up} sk'$ corresponding to pk'. The essential promise of UPKE is that ciphertexts encrypted to pk remain secure even when an adversary learns pk, the update token up and the updated *secret* key sk'. Thus, a protocol in which parties update receivers' key pairs whenever

[3] Note that new keys cannot be prepared and distributed too far in advance since this only extends the window of time during which forward secrecy is *not* provided.

encrypting to them can achieve relatively strong forward secrecy properties. Indeed, done right, no secret key is ever used more than once by a party and is immediately deleted (and replaced) upon first use.

However, there is a caveat to this. While using UPKE this way doesn't require as much coordination between parties as the naïve approach, it does require more than PPKE. To ensure a receiver has the correct secret key available, a sender must encrypt to the *most recent version* of the receiver's public key. In other words, senders must see each others' up tokens (or at least the most recently updated public key) before they can send. Otherwise, two senders may concurrently produce update tokens up_0 and up_1 for one public key pk giving rise to two sibling key pairs (pk_0, sk_0) and (pk_1, sk_1). We refer to this as a "fork". When a fork occurs, a receiver will typically only derive one of the forked secret keys sk_b since it must then immediately delete sk to ensure forward security. Thus, when it later receives up_{1-b}, it can no longer produce sk_{1-b} meaning it can't decrypt anything sent to pk_{1-b} (or any of its descendent keys). A similar restriction is that the receiver must decrypt ciphertexts in the same order they were sent (even when sent by different senders).

Still, compared to the naïve technique this represents a qualitative reduction in coordination since the receiver can essentially stay silent after initial public key distribution. Crucially, this makes asynchronous communication (as understood in asynchronous (group) messaging) possible, because senders need not wait for a receiver to announce new public keys before they can encrypt new messages to them. Thus, UPKE provides to secure messaging protocol designers the benefits of strong forward secrecy without forcing them to compromise on the ability of parties to privately message each other despite receivers potentially being off-line for extended periods of time.

Unfortunately, no UPKE scheme to date is sufficiently flexible, nor has all of the requisite security properties for natural use in CGKA and SGM applications which UPKE was partly designed for. Indeed, the initial academic work [ACDT20] in this area introduced rTreeKEM, a CGKA protocol which provides strong forward security by using UPKE in place of the PKE. The goal of [ACDT20] was to provide a more secure CGKA upon which to re-base the IETF's Messaging Layer Security (MLS) protocol, an open SGM standard specified in RFC9420 [BBR+23]. However, rTreeKEM (and the resulting SGM based on rTreeKEM [ACDT21]) were only analyzed in a relatively restricted model, which lead to relatively lightweight demands being placed on the underlying UPKE (both in terms of functionality and security).

Since then, however, the much more realistic "insider security" paradigm [AJM22] has established itself as a standard in the CGKA and SGM literature [HKP+21, AHKM22, AMT23]. Unlike the security models of [ACDT20, ACDT21], which assume authenticated channels, insider security only uses an insecure network. More challengingly maybe, insider security also provides meaningful security guarantees to parties joining "fake" groups; that is, sessions created arbitrarily by the adversary. These additions mean that insider security better captures the practical security concerns for SGM and CGKA. However, they also mean that to date, all UPKE schemes lack either the flexibility or security necessary for a CGKA (or SGM) application like rTreeKEM to be insider-secure.

Fake-group security. One such missing security property of existing UPKE notions is the (intuitive) property we call "joiner" security. When UPKE is used in higher-level CGKA/SGM protocols as a forward-secure replacement for PKE (as in rTreeKEM, for example), the joiner security of the UPKE scheme plays a central role in ensuring that the resulting CGKA/SGM protocol provides the "fake group" security aspect of insider security.

In more detail, CGKA and SGM protocols allow for dynamic groups (i.e. groups with evolving membership). Thus, a party P might receive an invitation to join an existing group mid-session. To join the group, P also receives the group state including the signature verification keys for each group member (authenticated by some trusted PKI). Fake-group security (for SGM) considers the case when the invitation (and accompanying group state) were produced maliciously by the adversary (who may also corrupt parties). It mandates that if P validates the invitation and state (as specified by the protocol) and subsequently proceeds with the execution to a point where no corrupt signing keys are left in the group's state, then the session should return to a secure state. For example, P's messages to the group should remain hidden from the adversary. Notably, this should be the case even though the group state could still include (U)PKE keys obtained by P from the adversary.

Fake-group security in MLS. To date, the only protocol we are aware of that achieves fake-group security is MLS. It does so by including signatures in the public group state, which give P a way to identify which PKE keys in the state were (supposedly) generated by which party and to whom the party sent the decryption keys as part of the protocol execution. Whenever a party is removed from the group, so too are any keys they either (supposedly) generated or were sent. In the insider corruption model, leaking a party's signing key also leaks all other secret keys it knows. Thus, if the group ever reaches a state where only secure verification keys remain in the group state, we can conclude that all remaining public keys were generated by and sent to uncorrupted parties. As a result, under those conditions, MLS can provide P with meaningful security guarantees for the session.

UPKE breaks MLS's fake-group security mechanism. When [ACDT20] proposed replacing PKE with UPKE to improve MLS's forward security, the authors left as an open problem how to adapt MLS's mechanism for fake-group security accordingly (at least without growing the group state in the number of updates to UPKE keys). This was one of the primary barriers to adopting UPKE in MLS.

Indeed, in general, the state of a group mid-session would include UPKE keys pk that are (nominally) the result of updates to some prior original key pk_0. So, to guarantee that pk is still secure, a new member must validate that (i) pk_0 was generated by an honest party, and (ii) that pk is the result of honestly using the update algorithm starting from pk_0.

One approach to providing (ii) could be to include in the group state all update tokens up leading from pk_0 to pk along with proofs that they were generated by the update algorithm. But this solution results in a state size and computational cost of joining that grow linearly in the number of updates between pk_0 and pk, which is prohibitive in practice. (MLS sessions can be expected to last for years and have, say, $n = 50,000$ group members; so it is not unrealistic that some of the $2n$ public keys in an MLS state will have been updated $n/2$ times by the time a new member joins.) It is also not an adequate solution to have receivers (i.e., members who can compute the updated sk, which could be as few as a single party) sign the updated pk to attest to its correctness, as it conflicts with the asynchronous nature of MLS.[4]

This motivates the *joiner security* property of UPKE identified in this work. It provides a joiner P with a concise tag for validating that some UPKE public key pk is the result of an (unknown) sequence of honest updates to a given "origin" UPKE public key pk_0. Thus,

[4] Indeed, after an update by one group member, new members could only join the group after a different (receiving) group member comes online to validate and sign the updated key. This would mean that at least 2 existing group members are needed to invite a new member to the group.

if an uncorrupted honest party attests to having generated pk_0 via a signature (just as with the PKE keys in MLS) then we can again conclude, in the insider security model, that pk must be secure.

Our proposal: UPKE allowing for fake-group security. These issues show that there seems to be no easy way to efficiently adapt MLS's fake-group security mechanism to UPKE. So instead, we ask the UPKE scheme to directly provide a comparable public key validation mechanism for new members (and a matching security guarantee). A *joiner-secure* UPKE scheme thus includes an algorithm $\mathsf{Verify_{jt}}$ with 3 inputs: (i) a UPKE public key pk to be validated, (ii) an original public key pk_0 and (iii) a "joiner tag" jt. The tag must be constant-size, in particular, independent of how many updates might have lead to from pk_0 to pk.

The UPKE security game chooses the initial pk_0 honestly at the start of the game (reflecting that in the application we only expect security from pk if an honest party attested to having generated pk_0, e.g. via a signature). Then, the UPKE adversary may update pk_0 with honest (i.e., generated by the challenger) or potentially malicious tokens up. The adversary wins if it can come up with pk^* and jt^* which pass $\mathsf{Verify_{jt}}$ and for which it can break privacy (IND-CCA) of a ciphertext c^* encrypted to pk^*. However, the adversary loses if it corrupts a secret key created before requesting c^*.

This restriction excludes trivial attacks in which pk^* is an updated version of a corrupted key. On the other hand, the restriction is not tight in the sense that it also excludes corruptions that do not lead to trivial attacks. We believe that our joiner security is a good compromise for the following reasons. First, defining UPKE security that only excludes trivial attacks would require UPKE schemes with additional functionality, which seems to require inefficient constructions.[5] Second, our joiner security is sufficient to prove that MLS with UPKE achieves the same fake-group security as today's MLS with PKE. In fact, the above can be proven even using UPKE joiner security with *no corruptions at all.* This means that our joiner security notion with corruption could enable an even stronger flavor of fake-group security for MLS with UPKE. Indeed, in Section 7.2 we give an example of an MLS execution where MLS with UPKE satisfying our stronger joiner security is secure, but would not be so if its UPKE only satisfied a notion disallowing corruptions. Such a stronger notion for MLS has not been defined yet, and we leave this as an interesting open problem.

State of the art. No UPKE definition in the literature accounts for joiner security. For convenience, in Appendix B we provide the state-of-the-art security definition of [APS23], which is the updatable KEM adaptation of the UPKE notion of [DKW21], subsequently also used in [ALP22, APS23, AW23].

UPKE taxonomy. Hiding beneath the term "UPKE" and the high-level intuition above, we actually find a series of concrete schemes in the literature (e.g. [JMM19, ACJM20, EJKM22, ALP22, DKW21, AMT23, AW23, APS23]) that differ in their syntax, security properties and even the purposes they serve in the applications they were conceived for. To better interpret the results in our work, it is instructive to categorize these differences.

Long vs. short syntax: The most obvious differences between UPKE schemes are their various syntaxes. UPKE was first introduced in [JMM19] using an *(asymmetric) long* syntax also used in [AAN$^+$22, EJKM22]. Here, "long syntax" means that key updates are

[5] Essentially, the challenger needs some way to identify which pk's are old versions of pk^* provided by the adversary. This seems to require storing the whole update history in pk^* or jt^*.

generated and applied using stand-alone purpose-built algorithms. In contrast, in this work (as in [ACDT20, ACJM20, ACDT21]) we use a *short syntax*, where keys are updated as a side-effect of encryption and decryption, thereby obviating the need for explicit update tokens and associated algorithms. We opted for the simpler syntax as it suffices for the dynamic group protocol applications we focus on and converting to long syntax is trivial.

Further, the work [EJKM22] defines two variants of a long syntax. "Asymmetric" long syntax means an update $up = (pu, su)$ includes a public component pu for updating public keys and a private component su for updating corresponding secret keys. "Symmetric" long syntax uses a single value to update both public and private keys. The notions in [DKW21, ALP22, AW23, APS23] can be viewed as having a symmetric long syntax where the random coins used by the public key update algorithm are also the update token used for the private key.

CPA vs. CCA: The first UPKE applications needed only CPA-style UPKE as they either included additional mechanisms reducing the role of UPKE in their protocol [JMM19, AAN+22] or their application was analyzed in a model that disables all attacks that might leverage honest parties as decryption oracles. (For example, the use of ideal authenticated channels in [ACDT20] trivially prevents the adversary from injecting ciphertexts to honest parties.) However, subsequently, the stronger and more realistic "insider security" model [AJM22] has become the standard in the field [HKP+21, AHKM22, AMT23]. This motivated the need for CCA-style UPKE. Indeed, all subsequent UPKE constructions (including in this work) are now regularly proven secure with CCA-style security games.

Forking security: Almost all UPKE applications in the group setting involve multiple parties using the same UPKE secret key. An adversary that, say, controls the network can easily cause such parties to have diverging views of a protocol session's transcript. This can result in forked UPKE keys (i.e., the initial key is updated using different sequences of updates). Thus, for such settings UPKE schemes must provide security in the face of forks. To date, we know of no (explicitly defined) UPKE scheme with this property, including those in [JMM19, EJKM22, ALP22, AW23, DKW21, APS23] making them, a priori, insufficient for such applications.[6]

Notable exceptions are the schemes of [ACJM20, AMT23] that are (implicitly) based on hierarchical identity-based encryption (HIBE). Unfortunately, owing to their use of unbounded-depth HIBE, these are decidedly impractical for real-world applications leaving the state of UPKE for the group setting unsatisfactory.

Decryption oracles for old keys: Even assuming there are no forks, in a setting with multiple parties using the same UPKE secret key, one has to account for parties not seeing some of the updates (yet) and hence holding old versions of the secret key. Accordingly, UPKE security notions should account for the attacker trying to inject ciphertexts to such parties. More precisely, assume we want to prove that an SGM scheme using UPKE is secure against adversaries who can inject ciphertexts but can *not* create forks. Even this weaker notion requires a UPKE security notion where, even after receiving the challenge ciphertext, the adversary can use the decryption oracle for any old secret key. However, this is not covered by any CCA-style UPKE definition we know of, in particular, not for [AW23, DKW21, ALP22, APS23].

[6] This seems to have happened because initial applications of UPKE are either in the 2-party setting, where forking is inherently not possible [JMM19] or they used very restricted models that artificially avoided forking by definition. Later UPKE constructions relied on UPKE security notions inspired by these early works but were not analyzed in their motivating applications using newer models. We provide a concrete scheme in Appendix A satisfying the definition [DKW21] but which leads to simple attacks when plugged into rTreeKEM.

Table 1: Comparison of security properties of different UPKE schemes. The last two columns indicates whether their constructions are practically efficient and in which model they are proven secure. AGM stands for the algebraic group model [FKL18].

Scheme	Syntax	Privacy	Forking	Agnostic	Update Validation	Joiner Security	PQ	Practical	Model
[JS18]	long	CCA		✓			✓		ROM
[PR18]	long	CCA		✓			✓		ROM
[JMM19]	long	CPA		✓				✓	ROM
[ACDT20]	short	CPA		✓				✓	ROM
[EJKM22]	long	CPA		✓			✓	✓	standard
[DKW21]	long	CCA			✓		✓		standard
[ALP22]	long	CCA			✓			✓	ROM
[AW23]	long	CCA					✓		ROM
[APS23]	long	CCA			✓		✓	✓	ROM
[ACJM20]	long	CCA	✓	✓	✓		✓		standard
[AMT23]	long	CCA	✓	✓	✓		✓		standard
This Work	short	CCA	✓		✓	✓		✓	ROM+AGM

Agnostic updates: The applications of UPKE considered in [JS18, PR18, JMM19, AAN$^+$22] require update tokens to be generated without knowing the public key to which they will ultimately be applied which we refer to as "agnostic" updates. Consequently, the UPKE schemes in those works are agnostic (as is the one in [EJKM22] and the implicit ones in [ACJM20, AMT23], although this is not necessary for the applications in those works). Conversely, the constructions of [AW23, DKW21, ALP22, APS23] create updates for a target key.

Protocol usage: While UPKE is usually billed and used as a tool for achieving forward security in an application, the work of [AAN$^+$22] is an exception. There, applying honestly generated updates to a possibly leaked secret key should refresh it to a new *secure* secret key. In other words (in addition to forward security), their protocol also relies on UPKE updates to ensure *post-compromise security* (PCS).[7] Thus, unlike any other use for UPKE we are aware of, [AAN$^+$22] needs the additional intuitive property that secret keys of updated public keys have high (computational) entropy given the old secret key and updated public key. Fortunately, to the best of our knowledge, most UPKE schemes already have this property with the exception of the HIBE-based implicit schemes in [JS18, PR18, ACJM20, AMT23]. For the purpose of this work we focus on using UPKE for forward secrecy, so we leave such an entropy requirement for future work.

Publicly verifiable updates: In multi-party protocols like MLS and rTreeKEM, a common feature is that more than one user might encrypt messages to a particular public key. Suppose we use UPKE in this setting and a party P_1 updates a public key pk to pk'. It is important that everyone in the group is convinced that pk' was generated via an honest update. Otherwise, a corrupt group member P_1 (called an *insider*) might generate a key pair (pk^*, sk^*) using KeyGen and then convince someone that pk^* is the updated key. Clearly this would be a problem as it would make all future ciphertext sent to the "updated key" pk^* decryptable by P_1.

For group members that know sk, avoiding this is usually not too difficult. For example, P_1 could encrypt to pk the coins used to produce the update [ACDT20]. However, revealing those coins to members who do *not* know sk would be problematic since UPKE security

[7] PCS is the mirror image of forward security where *future* keys should be secure despite *past* compromises.

notions only ensure forward secrecy for updated keys if the coins used to update sk to sk' are kept secret.

So, to prevent an insider from tricking parties that don't know sk into accepting arbitrary new public keys, the UPKE scheme should provide a method to publicly verify that pk' was produced from pk via the update algorithm. To achieve this, the verification procedure can also take as input a *validation tag* provided by P_1 as part of the message it sends to the group to announce the update. Intuitively, UPKE security should guarantee that if pk is secure and the pair (pk, pk') passes validation (with some tag), then pk' is also secure. Accordingly, the UPKE constructions [ALP22, DKW21, APS23] include a special $\mathsf{VerifyUpdate}$ algorithm. For the implicit HIBE-based schemes of [ACJM20, AMT23], update verification is quite trivial and the step is left implicit.

To summarize, no UPKE scheme to date is known to satisfy the (CCA and) forking security properties needed to use UPKE in a CGKA protocol like rTreeKEM [ACDT20] and meet the standard insider security for CGKA. (See Appendix A for a toy scheme that satisfies the UPKE security notion of [DKW21, ALP22, APS23], yet leads to a trivial insider security attack when used in place of PKE in MLS as proposed in [ACDT20]. The attack leverages the lack of forking security in those UPKE notions.)

Our Contributions

New model. In this work, we study CCA-secure Updatable Key Encapsulation Mechanisms (UKEM); the KEM analogue of UPKE. Note that building UPKE from a UKEM is straightforward (for both the long and short syntax) e.g. using a standard KEM/DEM construction of CCA-secure PKE from a CCA-secure KEM and a CCA-secure authenticated encryption scheme, as done for example in Hybrid Public Key Encryption (HPKE) [BBLW22].

We present a new UKEM syntax and security definition designed to meet the needs of dynamic group protocols such as MLS and rTreeKEM of [ACDT20]. In particular, it captures CCA-type confidentiality with forks and joiner security. Our notion for UKEM can be easily extended to model UPKE security.

The new syntax does not require agnostic updates as this is not needed for these applications. It is based on the short UPKE syntax augmented with two public key validation algorithms. The first, $\mathsf{Verify}_{\mathsf{jt}}$, lets new members joining a group validate the public keys they download as part of the group's state. It takes as input a public key pk_0 sampled via key generation, a public key pk_i being validated and a *joiner tag* jt_i. The joiner tag for a key pk_i is generated along with pk_i. In particular, the tag jt_0 is generated alongside pk_0 by KeyGen and for $i > 0$, the tag jt_i is generated together with pk_i by Encaps when encrypting to and updating pk_{i-1}, given only pk_{i-1} and jt_{i-1}.

Joiner tags can be used to provide new-member security in protocols like MLS and rTreeKEM as follows. In addition to each UPKE public key pk_i, the group state contains the associated tag jt_i, as well as the original key pk_0 signed by the group member who generated it.[8] Whenever a group member encrypts to pk_{i-1}, they replace pk_{i-1} and jt_{i-1} by pk_i and jt_i. Note that this can be done by all members, including new ones who did not see $pk_1, \ldots, pk_{i-2}$. Further, new members can verify the signature on pk_0 and verify jt_i, which convinces them, respectively, that pk_0 was honestly generated and then updated to get pk_i.

[8] The number of signatures can be reduced by half using the same "hashing down the path" optimization as in the parent hash mechanism of MLS.

The second algorithm, $\mathsf{Verify_{mt}}$ plays the same role as $\mathsf{VerifyUpdate}$ in the syntax of [ALP22, DKW21, APS23]. It allows existing group members that do not know the secret keys to validate an updated public key. It takes as input the previous public key pk_{i-1}, the updated public key pk_i and a member tag mt_i, also produced as part of the output when encapsulating to pk_{i-1}.

One may wonder why $\mathsf{Verify_{mt}}$ is needed and why members cannot verify $\mathsf{Verify_{jt}}$ instead. Indeed, there may exist schemes for which this is the case. However, constructing $\mathsf{Verify_{mt}}$ is much easier. Intuitively, this is because the creator of mt_i can use the actual "witness" (i.e., secret randomness) for updating pk_{i-1} to pk_i. On the other hand, jt_i must be generated without knowledge of the witnesses of the updates from pk_0 up to pk_{i-1}. As a result, our efficient construction achieves better security for $\mathsf{Verify_{mt}}$. On the other hand, joiners cannot profit from this additional security.

Our construction. We provide a practically efficient construction of UKEM satisfying our model based on pairing-friendly elliptic curves. We prove it secure in the combination of the random oracle model (ROM) and the algebraic group model (AGM) [FKL18] (see below) under the co-discrete-log assumption for bilinear groups, which in the AGM directly implies the co-CDH assumption [BLS01].[9]

Our starting point is the ElGamal-based KEM of DHIES [ABR98]. Public keys are of the form $u = g^x \in \mathbb{G}$ in a group $\mathbb{G}$ of prime order p with secret key $x \in \mathbb{Z}_p$. To encapsulate a symmetric key K, one chooses $r \leftarrow\$ \mathbb{Z}_p$, computes the ciphertext $v := g^r$ and sets $K := H(u, u^r)$, where H is treated as a random oracle.

To update a public key u in our scheme, we choose a random $d \leftarrow\$ \mathbb{Z}_p$, which defines a new key $u' := u \cdot g^d$. The associated member tag mt is a *proof of knowledge* (PoK) of d. Intuitively, this proof guarantees that if u was "secure" then so is u'. Indeed, suppose an adversary could update a random key $u = g^x$ to $u' = g^y$ for which it knows the secret key y while also proving knowledge of d such that $u' = u \cdot g^d$. Then by extracting d from the PoK we can use the adversary to compute the discrete log $x = y - d$ for a random u. For our scheme, this intuition about the one-wayness of u and u' also extends to CCA-security. To allow receivers to update their secret keys accordingly, d is encrypted under u. Decrypters can thus recover d and update secret key x to $x' := x + d$ for u'.

In fact, in our construction, d is actually derived via a random oracle (like the encapsulated key K). This achieves three goals. First, it allows us to deal with adaptive corruptions, a problem resulting from forks (see below). Second, unlike in [JMM19, ACDT20], we can use the KEM ciphertext directly to transmit d, which saves on encrypting d explicitly. Third, using encryption would require key-dependent message security.

Our UKEM *member security* notion requires CCA-security for any public key whose member tag is valid under $\mathsf{Verify_{mt}}$. The notion is strong in that it allows the adversary to adaptively corrupt any secret key sk as long as sk does not let the adversary learn the challenge secret key in a trivial way. We achieve this by leveraging the random oracle and by devising a careful guessing strategy: the security reduction guesses the first key u^* on the path of key updates leading from an initial honestly generated public key u_0 to the challenge public key for which (i) the adversary breaks an encryption (which, as in DHIES, corresponds to solving CDH) or (ii) it breaks an encryption of any key the adversary derived from u^*. Note that the reduction does not know this path and so it simply guesses a key.

[9] The co-DL assumption in groups $\mathbb{G}$ and $\hat{\mathbb{G}}$, both of prime order p and generated by g and h, respectively, states that given $g^x \in \mathbb{G}$ and $h^x \in \hat{\mathbb{G}}$ for $x \leftarrow\$ \mathbb{Z}_p$, it is hard to compute x. The co-CDH assumption states that given (g^x, g^r, h^x) for $x, r \leftarrow\$ \mathbb{Z}_p$, it is hard to compute g^{xr}.

Despite allowing adaptive corruption, our reduction achieves a security loss of only the number of ciphertexts (and thus new keys) the adversary asks for. For this to work, we need to assume that the proofs of knowledge of d (i.e., the member tags mt) are *simulation-sound*, that is, even after the adversary has seen simulated proofs (which the reduction creates when embedding its CDH challenge as a key), we can extract from an adversarial proof mt. This lets us "translate" a CDH solution for a key the adversary derived from the embedded key u^* to a solution for u^*.

Aiming for efficiency, we instantiate these proofs of knowledge of logarithms with Schnorr proofs, which consist of one element from $\mathbb{G}$ and one from $\mathbb{Z}_p$. These proofs were shown simulation-sound in the ROM and the algebraic group model [FO22], which provides "straight-line extractability". That is, extraction of the witness does not require rewinding the adversary (as in the security proof in the ROM [PS00]), which means we can extract from several proofs without risking an explosion of the running time due to interleaved rewinds for several proofs.

Joiner security. A trivial construction of a joiner tag jt would be to include all mt proofs and intermediary public keys on the path from u_0 to u', which guarantee knowledge of $d_1, \ldots, d_k$ s.t. $u' = u_0 \cdot g^d$ for $d = d_1 + \cdots + d_k$. However, this is inefficient and our goal is constant-size joiner tags. Since the updater does not know the value d, we need a way to "aggregate" the proofs mt_i guaranteeing honest hops from u_{i-1} to u_i into a single short proof jt guaranteeing honest hops from u_0 all the way to u'. An inherent problem with aggregatable proofs is that aggregation introduces malleability, which conflicts with our requirement that mt should be simulation-sound. Thus, we cannot hope that an instantiation of jt can also play the role of mt.

A very simple proof of knowledge of a logarithm is to assume that there exists a second generator h of $\mathbb{G}$ of which no one knows the discrete log. To prove knowledge of the logarithm of $v = g^d$, one sets $\pi := h^d$. The knowledge-of-exponent assumption [Dam92] states that π can only be computed if one knows d; formally, for any algorithm outputting (g^d, h^d), there exists an extractor that outputs d. These proofs can be efficiently aggregated: given a proof $\pi = h^d$ for $u = g^x$ w.r.t. $u_0 = g^{x_0}$, that is, $d = x - x_0$, a proof for $u' := u \cdot g^d$ is easily computed as $\pi' := \pi \cdot h^d$.

The problem is that, a priori, one cannot verify whether π was correctly computed. We thus embed our scheme in a *bilinear group*. That is, we assume a second group $\hat{\mathbb{G}}$ and a bilinear map $\mathrm{e}\colon \mathbb{G} \times \hat{\mathbb{G}} \to \mathbb{G}_T$ for some target group $\mathbb{G}_T$.[10] We can now set the basis h for the proofs as a generator of $\hat{\mathbb{G}}$ and use the pairing to verify a proof $\pi \in \hat{\mathbb{G}}$ for $v \in \mathbb{G}$ by checking whether $\mathrm{e}(v, h) = \mathrm{e}(g, \pi)$.

We prove joiner security directly in the algebraic group model. This model implies that after having received elements $h, \pi_1, \ldots, \pi_k \in \hat{\mathbb{G}}$, whenever the adversary returns some $\pi \in \hat{\mathbb{G}}$, it must have computed π as a linear combination ("algebraically") of all the $\hat{\mathbb{G}}$ elements it has received. In particular, the AGM assumes that the adversary outputs $\alpha_0, \ldots, \alpha_k \in \mathbb{Z}_p$ such that $\pi = h^{\alpha_0} \cdot \pi_1^{\alpha_1} \cdots \pi_k^{\alpha_k}$. In our security proof, h and the proofs $\pi_1, \ldots, \pi_k$ computed by the reduction will be all $\hat{\mathbb{G}}$ elements given to the adversary. As the reduction knows the discrete logarithms of the π_i's, it can compute the logarithm of π from $\alpha_0, \ldots, \alpha_k$.

Weaker assumption for member security. It turns out that the proofs π for joiner security also allow us to prove member security of our construction under standard PoK security

[10] In particular, we use an *asymmetric* pairing. That is, there are no efficiently computable homomorphisms between $\mathbb{G}$ and $\hat{\mathbb{G}}$. In practice, this type of pairing yields the most efficient constructions. Note also that assuming a pairing lets one prove the security of DHIES from co-CDH instead of the interactive assumption *gap-CDH* [OP01, ABR01] which is also the case for our UKEM (see below).

Table 2: Comparison of object sizes in {kilo, mega}-bytes of recent UPKE schemes. By ϕ we denote the bit-length of a NIZK that the update was generated correctly. A similar NIZK is needed to make the CPA scheme [DKW21] CCA-secure, while the CRS for the NIZK is included in public keys. In all UPKE applications considered in this work (e.g. rTreeKEM and MLS) ciphertexts are always sent together with a public key, an update *up*, joiner tag *jt* and member tag *mt*.

Scheme	Security	PQ	ROM	$\lvert sk\rvert$	$\lvert pk\rvert$	$\lvert ctxt\rvert$	$\lvert up\rvert$	$\lvert jt\rvert$	$\lvert mt\rvert$
[DKW21]	CPA	✓		166 B	41 KB	41 KB	52.375 MB		
[APS23]	CCA	✓				1.8 KB	10.8 KB + ϕ		
[ALP22]	CCA		✓	589 B	1.15 KB	11.375 KB	13.125 KB		
[AW23]	CCA		✓	32 B	80 B	96 B	128 B		
This work	CCA		✓	32 B	48 B	48 B		96 B	80 B

and with a tighter security proof. In particular, we only require a notion of simulation-soundness for *mt* where extraction is done *after* all simulations. To better understand the issue, recall that a co-CDH instance consists of $u = g^x$, $v = g^r \in \mathbb{G}$ and $\hat{u} = h^x \in \hat{\mathbb{G}}$ and the goal is to compute $w = g^{xr}$. In the security proof of DHIES, the reduction embeds u as the public key and v as the ciphertext and searches for w among the random oracle queries made by the adversary. Using co-CDH (rather than CDH) the reduction can efficiently find $w = g^{xr}$ using the pairing e, by checking if $\mathrm{e}(v, \hat{u}) \stackrel{?}{=} \mathrm{e}(w, h)$. (This is in contrast to returning w from a randomly chosen random oracle query, which would entail a multiplicative security loss in the number of adversary's random oracle queries.)

Our reduction for UKEM embeds u as some (honestly updated) public key and v as some ciphertext it hopes the adversary breaks. However, v may not be created for u but for some $u' = u \cdot g^d$ derived from u by the adversary, who needs to provide proofs *mt* and *jt* for u'. The reduction thus searches the random oracle queries for a value $w' = g^{(x+d)r}$. It could do so by extracting d from the proof of knowledge *mt*. However, using $\pi = h^d$, it can directly check $\mathrm{e}(v, \hat{u} \cdot \pi) \stackrel{?}{=} \mathrm{e}(w', h)$ without extracting anything at all. Extraction of the value d is then only needed when a CDH solution is found (and the reduction stops): computing $w := w'/v^d = g^{(x+d)r}/g^{rd}$ yields the co-CDH solution g^{xr}.

Efficiency of our scheme. We describe the efficiency profile of our scheme when instantiated with the BLS12-381 curve [SKSW22, Bow], which is a concrete instance of a BLS curve [BLS04] with conjectured 128-bit security. It is equipped with an asymmetric pairing from source groups $\mathbb{G} \times \hat{\mathbb{G}}$ to target group $\mathbb{G}_T$. Elements of $\mathbb{G}$ and $\hat{\mathbb{G}}$ are of size 48 B and 96 B respectively and the group order p has 32 B. As NIZK we use a Schnorr proof of knowledge of the discrete log of elements in $\mathbb{G}$. This means that NIZK proofs are elements of $\mathbb{G} \times \mathbb{Z}_p$ and thus of length 48 B + 32 B = 80 B. Based on this, in our scheme, public keys are 48 B, ciphertexts are 48 B, joiner tags are 96 B and member tags are 80 B. As seen in Table 2 this represents a *very* significant improvement over all CCA-secure UPKE (and UKEM) schemes to date (despite the new scheme satisfying a considerably stronger security notion).

For example, using UPKE in rTreeKEM to achieve insider security involves sending multiple tuples of the form $(pk, ctxt, t)$ where t is either an update token *up* or a joiner and member tag pair (jt, mt), depending on which UPKE syntax is used and *ctxt* is a ciphertext under the previous key. The tuples of the new UPKE construction in this work are around 1% the size of those of [ALP22]. For other CCA-secure schemes with publicly verifiable updates, the tuples are orders of magnitude larger still (despite none of these schemes providing forking or joiner security like the new construction).

We note that in our scheme, neither key generation, encapsulation nor decapsulation use pairing operations. One pairing is computed during each of the public key validation algorithms (which is run by parties holding the secret key before decapsulation as well).

Outlook. In Section 6 we discuss extensions of our security model and efficiency improvements of the construction. In Section 7 we dive into details of the impact of using variants of UPKE, including ours and less secure ones from the literature, on the security of MLS.

The main open problem left by our work (cf. Table 1) is a *post-quantum*-secure UPKE scheme that provides both forking and joiner security.

2 Preliminaries

Bilinear groups. Our scheme will be defined over a bilinear group with an asymmetric pairing, that is, a tuple $(p, \mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T, g, h, \mathrm{e})$, where $\mathbb{G}$ and $\hat{\mathbb{G}}$ are groups of prime order p generated by g and h, respectively, and $\mathrm{e}\colon \mathbb{G} \times \hat{\mathbb{G}} \to \mathbb{G}_T$ is a non-degenerate (i.e., $\mathrm{e}(g, h)$ generates $\mathbb{G}_T$) bilinear map (i.e., for all $a, b \in \mathbb{Z}_p$: $\mathrm{e}(g^a, h^b) = \mathrm{e}(g, h)^{ab}$).

The security of our scheme relies on the hardness of the co-discrete-logarithm problem in bilinear groups, defined as follows. We also state co-CDH [BLS01].

Definition 1 (co-DL). *Let $\mathcal{G} = (p, \mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T, g, h, \mathrm{e})$ be a bilinear group. The advantage of an adversary $\mathcal{A}$ in solving the co-DL problem over $\mathcal{G}$ is defined as*

$$\mathsf{Adv}^{\mathsf{co\text{-}DL}}_{\mathcal{G}}(\mathcal{A}) := \Pr\left[y = x \,\middle|\, \begin{array}{c} x \leftarrow\!\!\$\, \mathbb{Z}_p, u \leftarrow g^x, \hat{u} \leftarrow h^x \\ y \leftarrow \mathcal{A}(u, \hat{u}) \end{array} \right].$$

Definition 2 (co-CDH). *Let $\mathcal{G} = (p, \mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T, g, h, \mathrm{e})$ be a bilinear group. The advantage of an adversary $\mathcal{A}$ in solving the co-CDH problem over $\mathcal{G}$ is defined as*

$$\mathsf{Adv}^{\mathsf{co\text{-}CDH}}_{\mathcal{G}}(\mathcal{A}) := \Pr\left[w = g^{xr} \,\middle|\, \begin{array}{c} x, r \leftarrow\!\!\$\, \mathbb{Z}_p \\ u \leftarrow g^x, \hat{u} \leftarrow h^x, v \leftarrow g^r \\ w \leftarrow \mathcal{A}(u, \hat{u}, v) \end{array} \right].$$

For any $u = g^x$, $v = g^r$, we denote a CDH solution $w = g^{xr}$ by $w = \mathrm{DH}(u, v)$.

The algebraic group model. We analyze our scheme in the algebraic group model (AGM) [FKL18], which assumes that an adversary is *algebraic*, meaning that it computes any group element it outputs as a linear combination of the group elements it was given. More precisely, if the adversary, given input $g := u_0, u_1, \ldots, u_k \in \mathbb{G}$, outputs a group element $v \in \mathbb{G}$, then it must have computed v as $v = u_0^{\alpha_0} \cdots u_k^{\alpha_k}$ for some $\alpha_0, \ldots, \alpha_k$. Formally, the AGM assumes that such coefficients α_i, i.e., the "representation" of v are output by the adversary. The following is implicit in [FKL18].

Lemma 1. *In the algebraic group model, co-DL tightly implies co-CDH. In particular, for any algebraic adversary $\mathcal{A}$ against co-CDH in $\mathcal{G}$, there exists $\mathcal{B}$ against co-DL in $\mathcal{G}$ with approximately the same running time as $\mathcal{A}$ s.t. $\mathsf{Adv}^{\mathsf{co\text{-}DL}}_{\mathcal{G}}(\mathcal{B}) \geq \mathsf{Adv}^{\mathsf{co\text{-}CDH}}_{\mathcal{G}}(\mathcal{A})$.*

Proof. $\mathcal{B}$, on input a co-DL challenge $(u = g^x, \hat{u} = h^x)$, samples $s \leftarrow\!\!\$\, \mathbb{Z}_p^*$ and runs $\mathcal{A}$ on the co-CDH challenge $(u, \hat{u}, v := u \cdot g^s)$. When $\mathcal{B}$ returns a solution $w = g^{x(x+s)}$, it accompanies it with a representation (α, β, γ) s.t. $w = g^\alpha u^\beta v^\gamma = g^{\alpha + x\beta + (x+s)\gamma}$ (where g, u and v are the elements of $\mathbb{G}$ that $\mathcal{A}$ has seen). Equating the two representations of $\log w$ yields $x^2 - (\beta + \gamma - s)x - (\alpha + s\gamma) \equiv 0 \pmod p$, which $\mathcal{B}$ solves for x and returns the solution x satisfying $u = g^x$ (which must exist). $\square$

Simulation-extractable zero-knowledge proofs. Our UKEM scheme uses a proof system PoL ("proof of logarithm") for statements of the form $\theta := (u, u')$ proving knowledge of a witness d s.t. $u'/u = g^d$. Formally, PoL may make use of a random oracle H and comprises the following algorithms: $\tau \leftarrow \mathsf{PoL.Prove}_H((u, u'), d)$ outputs a proof τ and $0/1 \leftarrow \mathsf{PoL.Verify}_H((u, u'), \tau)$ verifies τ.

We require two security notions: *Zero-knowledge* (in the random oracle model) means that the reduction, which can program the random oracle H, can create proofs τ_i for statements θ_i without knowing a witness, using an algorithm $\mathsf{PoL.Simulate}_H$. The programmed random oracle and simulated proofs are, together, indistinguishable from a fresh random oracle and proofs computed honestly via $\mathsf{PoL.Prove}_H$ using a witness. We denote by $\epsilon^{\mathrm{sim}}_{\mathsf{PoL},n}$ the simulation error of PoL when simulating at most n proofs.

Strong simulation extractability (sSE) is an adaptation of *strong simulation soundness* [Sah99] to proofs of knowledge [DP92]. It is defined via the following game: an adversary $\mathcal{A}$ has access to random oracle H and an oracle that, on input a statement θ_i of $\mathcal{A}$'s choice, returns a simulated proof τ_i (and programs H as needed). Eventually, $\mathcal{A}$ returns a statement/proof pair $(\theta^*, \tau^*) \notin \{(\theta_i, \tau_i)\}_i$. If τ^* is a valid proof for θ^* (using the final programmed version of H) then a witness for θ^* can be extracted from $\mathcal{A}$. (The notion is *strong* since after querying a simulated proof for a statement, a different proof for the same statement must be extractable.) We require a *multi-extraction* version of sSE, in which, after having queried simulated proofs, the adversary returns *several* valid pairs (θ^*_i, τ^*_i) with $\{(\theta^*_i, \tau^*_i)\}_i \cap \{(\theta_i, \tau_i)\}_i = \emptyset$ and one can extract witnesses for all statements θ^*_i. We denote by $\epsilon^{\mathrm{ext}}_{\mathsf{PoL},n}(\mathcal{A})$ the advantage of the adversary $\mathcal{A}$ in breaking simulation extractability of PoL when returning at most n proofs.

Schnorr signatures. (Key-prefixed) Schnorr signatures are defined over a group $\mathbb{G}$ of order p and a hash function $H\colon \{0,1\}^* \to \mathbb{Z}_p$, modeled as a random oracle. Using signing key $x \in \mathbb{Z}_p$, a signature on a message $m \in \{0,1\}^*$ is computed by sampling $r \leftarrow\!\!\$\ \mathbb{Z}_p$ and returning

$$(v := g^r, s := (r + cx) \bmod p) \quad \text{with} \quad c := H(v, g^x, m).$$

A signature (v, s) is valid for message m under public key $u = g^x$ iff $g^s = v \cdot u^c$ with $c = H(v, u, m)$.

In the combination of the random oracle model and the algebraic group model, [FO22] show that Schnorr signatures are sSE zero-knowledge proofs of knowledge of the logarithm of the public key. That is, they are proofs of knowledge (of the witness) for the NP-relation $\{((u, m), x) \mid u = g^x, m \in \{0,1\}^*\}$.

Proofs for statements (u_i, m_i) can be simulated by programming the random oracle (as done in the original security proof for Schnorr [PS00]). Suppose an algebraic adversary $\mathcal{A}$ receives simulated proofs (v_i, s_i) for statements (u_i, m_i) of its choosing and then outputs a valid statement/proof pair $((u^*, m^*), (v^*, s^*)) \notin \{((u_i, m_i), (v_i, s_i))\}$. Then, [FO22] showed that from the representations for the group elements $u_1, u_2, \ldots, u^*$ and v^*, which $\mathcal{A}$ outputted during the game, one can efficiently compute a witness for the statement (u^*, m^*) with overwhelming probability.[11] In particular, extraction is straight-line and

[11] One might wonder why extraction is not trivial in the AGM anyway: an algebraic adversary that has only seen the generator g and returns u^* must know a representation α s.t. $u^* = g^\alpha$. In the context of security proofs, this is not the case: Consider e.g., an *algebraic* reduction $\mathcal{R}$ to the DL problem. This means that $\mathcal{R}$ receives a DL instance g^* and simulates the game to an adversary $\mathcal{A}$, providing it with group elements it computes *as linear combinations* of g and g^*. When $\mathcal{A}$ outputs a group element z, it accompanies it by a representation in basis all group elements received from $\mathcal{R}$. From this, $\mathcal{R}$ can compute a representation (α_0, α_1) in basis (g, g^*), that is, $z = g^{\alpha_0} \cdot (g^*)^{\alpha_1}$. To argue that $\mathcal{R}$ can extract from proofs of knowledge made by $\mathcal{A}$, we need to turn $\mathcal{R}$ together with $\mathcal{A}$ into an adversary

we can extract witnesses for *multiple* proofs produced during a single execution of an adversary. Thus, Schnorr signatures are *multi-extraction* sSE proofs in the ROM and AGM, which we formally prove in Appendix C.

The proof system PoL for member tags is defined as taking input a statement (u, u') and a witness $d = \log(u'/u)$ and returning a Schnorr signature under key u'/u on the message (u, u'). Then, sSE guarantees that after receiving simulated proofs for pairs (u_i, u'_i), if the adversary returns a new valid statement/proof pair $((u_*, u'_*), (v_*, s_*))$, we can extract d such that $u'_*/u_* = g^d$.

3 Updatable Key Encapsulation (UKEM)

3.1 Functionality

Intuitively, a UKEM scheme is a key encapsulation mechanism with the following modifications. First, on input a public key pk_i, the Encaps algorithm outputs – in addition to the key K and the ciphertext c – the updated public key pk_{i+1}. Accordingly, on input sk_i, the Decaps algorithm outputs – in addition to K – the updated secret key sk_{i+1}. This is analogous to any UKEM/UPKE with short syntax from the literature.

Second, Encaps also outputs a "member tag" mt_{i+1} which can be used by entities holding pk_i to validate pk_{i+1}. In particular, running $\mathsf{Verify}_{\mathsf{mt}}(pk_i, pk_{i+1}, mt_{i+1})$, such entities can verify that if pk_i is "honest" then pk_{i+1} is so, too. In MLS (more precisely, rTreeKEM [ACDT20]), $\mathsf{Verify}_{\mathsf{mt}}$ is run by members (not joiners) who do not know sk_i but know and have validated pk_i.

Third, Encaps also generates a "joiner tag" jt_{i+1} which can be used by entities holding pk_0 to validate pk_{i+1}. In particular, running $\mathsf{Verify}_{\mathsf{jt}}(pk_0, pk_{i+1}, jt_{i+1})$, such entities can verify that if pk_0 is "honest" then pk_{i+1} is so, too. In MLS, $\mathsf{Verify}_{\mathsf{jt}}$ is run by joiners after checking that pk_0 was signed by the member who generated it using KeyGen. Moreover, Encaps takes the last joiner tag jt_i as input.

Decaps takes additional input pk_{i+1} and should output $\perp$ if it does not "match" sk_{i+1}. In MLS, members who *do* know sk_i can thus reject "incorrect" (e.g. adversarially chosen) pk_{i+1}.

Formally, a UKEM scheme consists of the following algorithms:

Key Generation. $\mathsf{KeyGen}(\kappa) \to (pk_0, sk_0, jt_0)$, on input the security parameter, outputs a key pair (pk_0, sk_0) and the first joiner tag jt_0.

Encapsulation. $\mathsf{Encaps}(pk_i, jt_i) \to (K, c, pk_{i+1}, mt_{i+1}, jt_{i+1})$ takes as input the current public key and joiner tag and returns an encapsulated key K, a ciphertext c, an updated public key pk_{i+1}, a new member tag mt_{i+1} and an updated joiner tag jt_{i+1}.

Verification of member tags. $\mathsf{Verify}_{\mathsf{mt}}(pk_i, pk_{i+1}, mt_{i+1}) \to 0/1$ verifies the update from pk_i to pk_{i+1} using the tag mt_{i+1}.

Verification of joiner tags. $\mathsf{Verify}_{\mathsf{jt}}(pk_0, pk_{i+1}, jt_{i+1}) \to 0/1$ verifies the update from pk_0 to pk_{i+1} using the tag jt_{i+1}.

Decapsulation. $\mathsf{Decaps}(sk_i, c, pk_{i+1}) \to (K, sk_{i+1})/\perp$ outputs the decapsulated key K and the updated secret key sk_{i+1}, but only if pk_{i+1} matches sk_{i+1}.

against simulation-extractability. This adversary is algebraic, but only in the sense that it can give representations in basis (g, g^*) where g^* is a group element of which the extractor will not know the discrete logarithm. Therefore, [FO22] (and our proof in Appendix C) actually show that even in the presence of an "auxiliary-input" g^*, one can extract the witness from a Schnorr proof.

Using UKEM schemes. Importantly, Decaps *does not validate any tags.* Therefore, applications using a UKEM scheme *should always run* $\mathsf{Verify}_{\mathsf{mt}}$ *and* $\mathsf{Verify}_{\mathsf{jt}}$ *before* Decaps. This is reflected in our security notion.

Correctness. A UKEM scheme is correct if for all $\ell \geq 1$ the probability of $\mathsf{Correct}$ is overwhelming, where $\mathsf{Correct}$ denotes the output of the following experiment. Generate $(pk_0, sk_0, jt_0) \leftarrow \mathsf{KeyGen}(\kappa)$. If $\mathsf{Verify}_{\mathsf{jt}}(pk_0, pk_0, jt_0) = 0$, output 0. For each $i \in [0, \ell - 1]$:

– Compute $(K, c, pk_{i+1}, mt_{i+1}, jt_{i+1}) \leftarrow \mathsf{Encaps}(pk_i, jt_i)$.
– If $\mathsf{Verify}_{\mathsf{mt}}(pk_i, pk_{i+1}, mt_{i+1}) = 0$, output 0.
– If $\mathsf{Verify}_{\mathsf{jt}}(pk_0, pk_{i+1}, jt_{i+1}) = 0$, output 0.
– Compute $(K', sk_{i+1}) \leftarrow \mathsf{Decaps}(sk_i, c, pk_{i+1})$. If $(K', sk_{i+1}) = \perp$ or $K' \neq K$, output 0.

Output 1.

3.2 Security

The $\mathsf{IND\text{-}CCA}$ security of UKEM schemes is formalized by the experiment in Figure 1. Intuitively, during the experiment, a tree is created where each node is identified by an integer i and has a public key pk_i and a joiner tag jt_i. The root is identified by $i = 0$. Each non-root node has a parent par_i and a member tag mt_i. Further, some nodes have a secret key sk_i. If a node has a secret key, we call it *full*, and otherwise we call it a *half node.*

The root node $i = 0$ is created by the challenger at the beginning of the experiment. Its public key pk_0, secret key sk_0 and joiner tag jt_0 are generated using KeyGen (the root is thus a full node). All other nodes j are created by updating existing nodes in one of three ways:

1. When the adversary $\mathcal{A}$ calls the oracle $\mathrm{Enc}(i)$, the challenger creates a child j of i by running Encaps. If i is a full node, j is also a full node with secret key generated by running Decaps. $\mathcal{A}$ is also given the generated ciphertext and key.
2. A child of i with a possibly "adversarial" public key may be created when $\mathcal{A}$ calls the oracle $\mathrm{Dec}(i, c, pk', mt', jt')$. In such case, the challenger verifies mt' and jt' and, if the check passes, creates the node j using these values. If i is a full node and $\mathsf{Decaps}(sk_i, c, pk')$ outputs (K, sk_j) (and not $\perp$), then j is also a full node with secret key sk_j; in that case, $\mathcal{A}$ also receives K, which reflects CCA-security. Otherwise, j is a half node. Observe that j is a half node if $\mathcal{A}$ provides correct (publicly verifiable) tags but c inconsistent with pk' (which is not publicly verifiable).
3. A node can be created during a challenge call. We address such calls next. There are two challenge oracles: member challenge MChal and joiner challenge JChal. Without loss of generality, $\mathcal{A}$ can only call one of them, and only once.

Member security. Consider the case that $\mathcal{A}$ calls MChal, which means that the notion implies security for group members when used in a secure messaging application. On query $\mathrm{MChal}(i^*)$, the challenger creates a child j^* of i^* just like during an Enc query creating a "real" key $K^{(1)}$. $\mathcal{A}$ gets either $K^{(1)}$ or a random and independent key $K^{(0)}$ and has to decide which is the case. It also receives the resulting tags, public key and the ciphertext c^*. To disable trivial wins, on inputs i and c the Dec oracle returns $\perp$ if $pk_i = pk_{i^*}$ and $c = c^*$.

Furthermore, our notion implies forward secrecy by giving $\mathcal{A}$ access to an oracle Rev, which reveals secret keys (of full nodes). In particular, $\mathcal{A}$ can ask for the secret key of any node outside the *challenge set of* i^*, which consists of three parts. First, the *base* of the challenge set, which is the path from the root 0 to i^*. Clearly, revealing the secret key

Game UKEM IND-CCA Security

```
$\mathbf{Exp}^{\mathsf{IND\text{-}CCA}}(\mathcal{A})$
  $(pk_0, sk_0, jt_0) \leftarrow \mathsf{KeyGen}(\kappa)$
  $(mt_0, par_0, rev_0, j) \leftarrow (\epsilon, \epsilon, 0, 0)$
  $b \leftarrow\$ \{0,1\}$
  $b' \leftarrow \mathcal{A}^{\mathsf{Enc},\mathsf{Dec},\mathsf{Rev},\mathsf{MChal},\mathsf{JChal}}(pk_0, jt_0)$
  $S \leftarrow \textsf{chall-set}(chall)$
  return $b = b' \wedge \forall j \in S : \neg rev_j$
```

```
Oracle $\mathsf{Enc}(i)$
  $(K, c) \leftarrow \textsf{create-honest-node}(i)$
  return $(K, c, pk_j, mt_j, jt_j)$
```

```
Oracle $\mathsf{MChal}(i)$
  req $chall = \perp$
  $K^{(0)} \leftarrow\$ \{0,1\}^\kappa$
  $(K^{(1)}, c^*) \leftarrow \textsf{create-honest-node}(i)$
  $chall \leftarrow (\text{“member”}, i, c^*, pk_i)$
  return $(K^{(b)}, c^*, pk_j, mt_j, jt_j)$
```

```
Oracle $\mathsf{Rev}(i)$
  req $sk_i \neq \perp$
  $rev_i \leftarrow 1$
  return $sk_i$
```

```
Oracle $\mathsf{JChal}(pk', jt')$
  req $chall = \perp$
  req $\mathsf{Verify}_{\mathsf{jt}}(pk_0, pk', jt')$
  $K^{(0)} \leftarrow\$ \{0,1\}^\kappa$
  $(K^{(1)}, c^*, pk, mt, jt) \leftarrow \mathsf{Encaps}(pk', jt')$
  $chall \leftarrow (\text{“joiner”}, j, c^*, pk')$
  return $(K^{(b)}, c^*, pk, mt, jt)$
```

```
Oracle $\mathsf{Dec}(i', c', pk', mt', jt')$
  req $pk_{i'} \neq \perp$ // $i$-th node exists
  if $chall = (*, *, c^*, pk^*)$ then
    req $c^* \neq c' \vee pk^* \neq pk_{i'}$
  req $\mathsf{Verify}_{\mathsf{mt}}(pk_{i'}, pk', mt')$
  req $\mathsf{Verify}_{\mathsf{jt}}(pk_0, pk', jt')$
  $j$++
  $(pk_j, mt_j, jt_j, sk_j, par_j, rev_j) \leftarrow (pk', mt', jt', \perp, i', 0)$
  if $sk_{i'} \neq \perp$ then
    $out \leftarrow \mathsf{Decaps}(sk_{i'}, c', pk')$
    if $out \neq \perp$ then
      $(K, sk_j) \leftarrow out$
      return $K$
  return $\perp$
```

```
Helper $\textsf{create-honest-node}(i)$
  req $pk_i \neq \perp$ // $i$-th node exists
  $j$++
  $(K, c, pk_j, mt_j, jt_j) \leftarrow \mathsf{Encaps}(pk_i, jt_i)$
  if $sk_i \neq \perp$ then
    // $i$-th node is full
    $(*, sk_j) \leftarrow \mathsf{Decaps}(sk_i, c, pk_j)$
  else $sk_j \leftarrow \perp$
  $(par_j, rev_j) \leftarrow (i, 0)$
  return $(K, c)$
```

```
Helper $\textsf{chall-set}(chall)$
  if $chall = (\text{“member”}, i^*, *, *)$ then
    $base \leftarrow \{i_0, \ldots, i_\ell\}$ where $i_0, \ldots, i_\ell$ is the path
    from node $i_0 = 0$ to node $i_\ell = i^*$
  else if $chall = (\text{“joiner”}, i^*, *, *)$ then
    $base \leftarrow \{0, \ldots, i^*\}$
  else return $\emptyset$
  $extd\text{-}base \leftarrow \{i' \mid \exists\, i \in base :$ // include duplicates
                   $(pk_{i'}, mt_{i'}, jt_{i'}) = (pk_i, mt_i, jt_i)\}$
  return $\textsf{dec-closure}(extd\text{-}base)$
```

```
Helper $\textsf{dec-closure}(S)$
  Return the set of all $j$ reachable from some $i \in S$
  via only edges created by Dec queries.
```

Fig. 1: The experiment formalizing UKEM IND-CCA security. By default, all variables are initialized to $\perp$. We use **req** *condition* to denote that if *condition* is false, then the current function, and any function calling it, stops and returns $\perp$.

for any such node would allow $\mathcal{A}$ to trivially win by computing the secret key of i^* by running Decaps sequentially on the ciphertexts between the corrupted and the challenged node, and then decapsulating c^*. This *base* is extended to *extd-base*, which also includes *duplicates*, i.e., any nodes that have the same public key and the same tags as a node in *base*.[12]

Finally, the challenge set contains *branches*, which are nodes reachable from *extd-base* via nodes created by Dec queries. This is where our notion does not formalize optimal security: there exist UKEM schemes, notably the ones based on HIBE that achieve security even when $\mathcal{A}$ can corrupt keys on branches. However, we are not aware of any *efficient* schemes that achieve this. Observe that the secret keys of nodes on branches are generated by updating a secret key on the challenge path (or a duplicate node) with updates generated by $\mathcal{A}$. Therefore, for optimal security we would need a mechanism that does not allow $\mathcal{A}$ to undo its updates, which resembles PPKE.

We note that $\mathcal{A}$ is allowed to ask for the secret key for j^* created by MChal, which corresponds to the fact that in typical UPKE security notions [DKW21, APS23, ALP22, AW23] the challenge oracle returns the updated secret key. However, $\mathcal{A}$ can also obtain many other keys, e.g., any node created by Enc and not on the challenge path (and all their children).

Joiner security. Next, consider the case that $\mathcal{A}$ calls JChal, formalizing a notion that implies security for joiners when used in a secure messaging application. On query JChal(pk', jt'), the challenger verifies jt' for pk' w.r.t. the (honest) pk_0 and, if the check passes, runs Encaps on pk' to generate the "real" key $K^{(1)}$. As for member security, $\mathcal{A}$ is also given the resulting ciphertext, public key and tags. $\mathcal{A}$'s goal is to distinguish $K^{(1)}$ from a random and independent $K^{(0)}$. To disable trivial wins, on inputs i and c the Dec oracle returns $\perp$ if $pk_i = pk'$ and $c = c^*$.

Reveal queries are more restricted for joiner security than for member security. In particular, the challenge set *base* now contains all nodes generated before the call to JChal was made (which is thus a superset of the set *base* in the MChal setting). Analogously to member security, $\mathcal{A}$ is not allowed to corrupt keys for nodes in the set *base*, any duplicates of such nodes and *branches* (i.e., nodes derived from these via Dec queries).

The above restriction cannot be relaxed without enabling "trivial" attacks against any correct scheme (at least with our syntax). To illustrate this, consider the following adversary $\mathcal{A}$. By calling Enc(0) twice, $\mathcal{A}$ generates two children of node 0 with keys pk_1, pk_2 and tags jt_1, jt_2. Then by running $\mathsf{Encaps}(pk_1, jt_1)$ (possibly repeating this to create a longer path), $\mathcal{A}$ computes a new pair (pk', jt') on its own and submits it to its JChal oracle. If $\mathcal{A}$ was allowed to query Rev(1), it could then, by running Decaps (possibly consecutively), compute the secret key for pk'.

In general, pk' may have been derived via Encaps from any pk_i that $\mathcal{A}$ saw before generating pk'. Our restriction thus disallows Rev(i) for all such pk_i, including pk_0, pk_1 and pk_2 in the above example, even though corrupting pk_2 would not lead to an attack. However, the challenger cannot identify keys that can be revealed, as the UKEM syntax does not allow to decide, given the challenger's information, whether pk' could *not* have been derived from them.

[12] This restriction prevents trivial attacks, as in the following example: $\mathcal{A}$ queries Enc(0), which creates node 1 with (pk_1, mt_1, jt_1) and ciphertext c_1. It next queries Rev(1), to obtain the corresponding sk_1. It then queries Dec$(0, c_1, pk_1, mt_1, jt_1)$, which creates node 2 with $sk_2 = sk_1$, and finally MChal(2), to receive $(c^*, K^*, pk_3, mt_3, jt_3)$ and checks whether for $(K', sk_3) \leftarrow \mathsf{Decaps}(sk_1, c^*, pk_3)$ it holds that $K' = K^*$.

Remark 1. One could consider relaxing the above restriction on reveal queries for a UPKE with modified syntax, e.g. with an additional algorithm that decides, given pk', jt', pk_i and sk_i (and any other information the challenger has), whether pk_i is an ancestor of pk'. However, implementing such an algorithm seems to require inefficient techniques, such as storing all ancestor public keys in jt'.

Remark 2. One could imagine strengthening joiner security by having $\mathrm{JChal}(pk', jt')$ create an (incomplete) node i' with $pk_{i'} = pk'$ and $jt_{i'} = jt'$ and allowing the adversary $\mathcal{A}$ to create a (detached) tree rooted at i'. (Note that, by the arguments in Remark 1, we cannot define a parent of i'.) However, the resulting notion would be equivalent to our notion. Since i' has no parent, its sub-tree contains only half-nodes without secret keys. So no oracle call related to such nodes uses any secrets unknown to $\mathcal{A}$ (which are the secret keys of full nodes and the bit b.) Thus, $\mathcal{A}$ could emulate such oracle calls itself.

Definition 3 (UKEM Security). *Let* $\mathrm{Exp}^{\mathsf{IND\text{-}CCA}}(\mathcal{A})$ *be as defined in Figure 1. The advantage of an adversary* $\mathcal{A}$ *against the* IND-CCA *security of a* UKEM *scheme is defined as*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}(\mathcal{A}) := 2\Pr\left[\mathrm{Exp}^{\mathsf{IND\text{-}CCA}}(\mathcal{A}) = 1\right] - 1.$$

4 Construction

The basis of our construction is the KEM part of DHIES [ABR98], which is basically "hashed ElGamal" for a hash function (modeled as a random oracle) $H\colon \{0,1\}^* \to \mathcal{K}$, the symmetric key space. We use groups $\mathbb{G}$ and $\hat{\mathbb{G}}$ of order p with a pairing e from $\mathbb{G} \times \hat{\mathbb{G}}$ and define the KEM in $\mathbb{G}$: Public keys are of the form $u = g^x \in \mathbb{G}$ and symmetric keys K are encapsulated by choosing $r \leftarrow\!\!\$\ \mathbb{Z}_p$, defining the ciphertext as $v := g^r$ and deriving $K := H(u, u^r)$. Using the secret key x, keys are decapsulated from v as $K := H(g^x, v^x)$.

We extend this to derive updated public keys as follows: using a second random oracle H_1, we define $d := H_1(u, u^r)$ and set the new public key as $u' := u \cdot g^d$. Decapsulation now takes as additional argument the updated key u', derives $d := H_1(g^x, v^x)$, updates the secret key to $x' := x + d$ and checks if $u' = g^{x'}$. To guarantee that the new key u' was derived correctly (and not chosen freshly with a known secret key), we add a proof of knowledge (PoK) τ of d, that is, a PoK of the discrete log of u'/u. (For our security notion allowing adaptive corruption, τ needs to be simulation-sound.) This τ corresponds to mt in the UKEM model.

The tag jt given to joiners will be a PoK of $D' := x' - x_0$, with x_0 the secret key of the root key u_0 and x' the secret key of the updated key u'. This guarantees that u' is linked to the root key u_0. A straightforward solution would be to define $jt_j := (u_1, mt_1, \ldots, u_{j-1}, mt_{j-1}, mt_j)$, that is, all keys between u_0 and u_j together with their update proofs. To avoid a growth in size depending on the number of updates, we would require a direct proof of knowledge of $D' = x' - x_0$, but the updater will not know D'. Our solution is to use "aggregatable" proofs, that is, given a PoK of $D = x - x_0$ corresponding to key u, and deriving u' from u using d, one should be able to derive a PoK of $D' := D + d$.

We use the second pairing source group $\hat{\mathbb{G}}$, generated by h, to instantiate these aggregatable proofs. A proof π proving knowledge of the logarithm of an element $u = g^x \in \mathbb{G}$ is defined as $\pi := h^x \in \hat{\mathbb{G}}$. Using the pairing, a proof can be verified by checking $\mathrm{e}(u, h) = \mathrm{e}(g, \pi)$. Making "knowledge-of-exponent"-type assumptions (in our security proof we will directly rely on the algebraic group model), we get that from any algorithm

Construction $\mathsf{UKEM}[H_1, H_2, H_3, \mathcal{G}, \mathsf{PoL}]$

```
KeyGen()
    $x \leftarrow\$ \mathbb{Z}_p$
    return $(u \leftarrow g^x, x, \pi_0 \leftarrow h^0)$

Encaps($u_i, \pi_i$)
    $r \leftarrow\$ \mathbb{Z}_p$
    $v \leftarrow g^r$ // ciphertext
    $d_{i+1} \leftarrow H_1(u_i, u_i^r)$ // secret key update
    $u_{i+1} \leftarrow u_i \cdot g^{d_{i+1}}$ // update public key
    $\tau_{i+1} \leftarrow \mathsf{PoL.Prove}_{H_3}((u_i, u_{i+1}), d_{i+1})$
    $\pi_{i+1} \leftarrow \pi_i \cdot h^{d_{i+1}}$
    $K \leftarrow H_2(u_i, u_i^r)$ // output key
    return $(K, v, u_{i+1}, \tau_{i+1}, \pi_{i+1})$

Verify_mt($u_i, u_{i+1}, \tau_{i+1}$)
    return $\mathsf{PoL.Verify}_{H_3}((u_i, u_{i+1}), \tau_{i+1})$

Verify_jt($u_0, u_{i+1}, \pi_{i+1}$)
    return $\mathrm{e}(u_{i+1}/u_0, h) = \mathrm{e}(g, \pi_{i+1})$

Decaps($x_i, v, u_{i+1}$)
    $d_{i+1} \leftarrow H_1(g^{x_i}, v^{x_i})$
    $x_{i+1} \leftarrow x_i + d_{i+1}$ // update secret key
    if $u_{i+1} \neq g^{x_{i+1}}$ then
        return $\perp$
    $K \leftarrow H_2(g^{x_i}, v^{x_i})$ // output symmetric key
    return $(K, x_{i+1})$
```

Fig. 2: The UKEM construction. Here H_1, H_2 and H_3 are hash functions modeled in the proof as random oracles, $\mathcal{G} = (p, \mathbb{G}, \hat{\mathbb{G}}, \mathbb{G}_T, g, h, \mathrm{e})$ is a bilinear group, and PoL is a proof of knowledge system for discrete logarithm statements in $\mathbb{G}$, which might use H_3.

that returns u and π satisfying the above equation, one can extract $x = \log_g u = \log_h \pi$, meaning π is indeed a proof of knowledge.

Using these proofs for *jt* allows the updater to transform a proof π for u into a proof $\pi' := \pi \cdot h^d$ for $u' = u \cdot g^d$. A proof π' for u' w.r.t. u_0 is verified by checking $\mathrm{e}(u'/u_0, h) = \mathrm{e}(g, \pi')$. Our UKEM scheme is formally defined in Figure 2.

5 Security of the Construction

Security of our construction is expressed by the following theorem.

Theorem 1. *If* PoL *is a strongly simulation-extractable proof system and co-CDH holds for $\mathcal{G}$, and assuming adversary $\mathcal{A}$ is algebraic, then the UKEM construction from Figure 2 is* IND-CCA *secure in the ROM. More precisely, for any adversary $\mathcal{A}$, there exist reductions $\mathcal{B}$ and $\mathcal{B}'$ such that*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}(\mathcal{A}) \leq (n_e + 2)\left(\epsilon^{\mathrm{sim}}_{\mathsf{PoL},n_e+1} + \epsilon^{\mathrm{ext}}_{\mathsf{PoL},n_d}(\mathcal{B}') + \mathsf{Adv}^{\mathsf{co\text{-}CDH}}_{\mathcal{G}}(\mathcal{B})\right),$$

where n_e (n_d, resp.) are upper bounds on the number of Enc (Dec, resp.) queries made by $\mathcal{A}$, and $\epsilon^{\mathrm{sim}}_{\mathsf{PoL},n}$ ($\epsilon^{\mathrm{ext}}_{\mathsf{PoL},n_d}(\cdot)$, resp.) are the probabilities that simulation of n proofs (extraction from n_d proofs, resp.) fails for PoL.

Together with Lemma 1, Theorem 1 implies that the security of our construction can be reduced to co-DL. Moreover, using the fact that Schnorr proofs, against algebraic adversaries, are strongly simulation-(multi-)extractable (as we show in Appendix C) with simulation error $\epsilon^{\mathrm{sim}}_n := n/(p - n_h - n)$ and (multi-)extraction error $\epsilon^{\mathrm{ext}}_n = n/p$, yields the following:

Corollary 1. *Let $\mathcal{G}$ be an asymmetric bilinear group. If* PoL *is instantiated using Schnorr (cf. Section 2) and co-DL holds for $\mathcal{G}$, then the UKEM construction from Figure 2 is* IND-CCA *secure in the ROM and the AGM. More precisely, for any algebraic adversary $\mathcal{A}$, there exist a reduction $\mathcal{B}$ such that*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}(\mathcal{A}) \leq (n_e + 2)\Big(\frac{n_e + 1}{p - n_h - n_e - 1} + \frac{n_d}{p} + \mathsf{Adv}^{\mathsf{co\text{-}DL}}_{\mathcal{G}}(\mathcal{B})\Big),$$

where n_e, n_d and n_h are upper bounds on the number of, respectively, Enc, Dec and H_3 queries made by $\mathcal{A}$.

Proof of Theorem 1. We split the security notion $\mathsf{IND\text{-}CCA}$ into two: $\mathsf{CCA\text{-}M}$, in which the JChal oracle is disabled, and $\mathsf{CCA\text{-}J}$, in which the MChal oracle is disabled. The advantages $\mathsf{Adv}^{\mathsf{CCA\text{-}M}}$ and $\mathsf{Adv}^{\mathsf{CCA\text{-}J}}$ are defined accordingly. In Lemmas 2 and 3 we then bound these advantages. Theorem 1 then follows by summing them and letting $\mathcal{B}$ and $\mathcal{B}'$ be those adversaries from Lemma 2 or Lemma 3 that have the greater advantage.

5.1 Member Security

We first prove the following lemma, which formalizes member security, $\mathsf{CCA\text{-}M}$, of our UKEM scheme.

Lemma 2. *If* PoL *is a strongly simulation-extractable proof system and co-CDH holds for* $\mathcal{G}$*, then the UKEM construction from Figure 2 is* $\mathsf{CCA\text{-}M}$*-secure in the ROM. More precisely, for any adversary* $\mathcal{A}$*, there exist reductions* $\mathcal{B}$ *and* $\mathcal{B}'$ *such that*

$$\mathsf{Adv}^{\mathsf{CCA\text{-}M}}(\mathcal{A}) \leq (n_e + 1)\big(\epsilon^{\mathrm{sim}}_{\mathsf{PoL},n_e+1} + \epsilon^{\mathrm{ext}}_{\mathsf{PoL},n_d}(\mathcal{B}') + \mathsf{Adv}^{\mathsf{co\text{-}CDH}}_{\mathcal{G}}(\mathcal{B})\big),$$

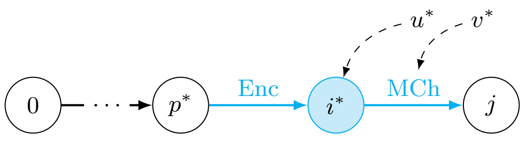
where n_e and n_d are upper bounds on the number of $\mathcal{A}$'s Enc and Dec queries, respectively.

Proof intuition. Let $\mathcal{A}$ be any adversary against the $\mathsf{CCA\text{-}M}$ security of our UKEM scheme. We will construct a reduction $\mathcal{B}$ against the co-CDH problem, i.e., given u^*, $\hat{u}^*$ and v^*, $\mathcal{B}$ must compute $w^* = \mathrm{DH}(u^*, v^*)$.

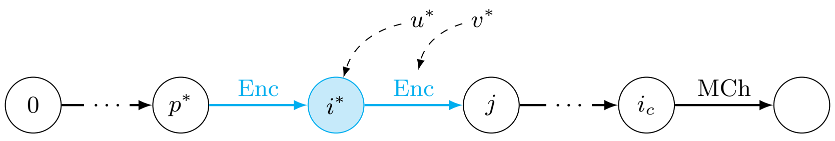
We start by adapting the proof idea for the security of the KEM of DHIES in the ROM. $\mathcal{B}$ embeds u^* as some u_j generated by the challenger, that is, either as u_0 or some u_j returned by an $\mathrm{Enc}(i)$ query, hoping that $\mathcal{A}$ calls $\mathrm{MChal}(j)$. If this happens, $\mathcal{B}$ embeds v^* as the ciphertext returned by the oracle. Now as long as $\mathcal{A}$ never queries (u^*, w^*) to the RO H_2 with $w^* = \mathrm{DH}(u^*, v^*)$, the challenge key $K^{(b)}$ is independently random in both the real and the ideal game, and so no information on b is revealed. On the other hand, querying (u^*, w^*) means $\mathcal{A}$ solved CDH; moreover, $\mathcal{B}$ can test this by checking if $\mathrm{e}(w^*, h) = \mathrm{e}(v^*, \hat{u}^*)$.

Embedding u^*. Consider embedding $u^* = g^x$ as u_{i^*} during a query $\mathrm{Enc}(p^*)$ (with p^* the parent of i^*), which returns ciphertext v_{i^*}. This is depicted in 3a. Recall that Encaps would compute $d_{i^*} = H_1(u_{p^*}, w_{i^*})$ with $w_{i^*} := \mathrm{DH}(u_{p^*}, v_{i^*})$ and define $u_{i^*} := u_{p^*} \cdot g^{d_{i^*}}$ and $\pi_{i^*} := \pi_{p^*} \cdot h^{d_{i^*}}$. So when setting $u_{i^*} := u^*$, the reduction $\mathcal{B}$ does not know the corresponding $d_{i^*} = \log(u^*/u_{p^*})$. It thus generates the proof τ_{i^*} using the simulator guaranteed by zero knowledge of PoL. To compute π_{i^*}, it uses $\hat{u}^* = h^x$ from its co-CDH challenge and sets $\pi_{i^*} := \hat{u}^*/h^{x_0}$ (and $\pi_{i^*} := h^0$ if $j = 0$).

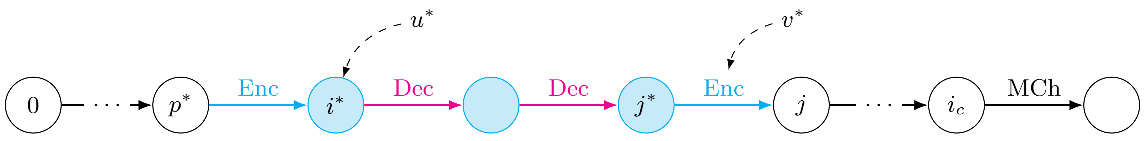
While $\mathcal{B}$ can simulate the proofs, not knowing d_{i^*}, it cannot consistently answer if $\mathcal{A}$ queries H_1 on (u_{p^*}, w_{i^*}). On the other hand, as long as this query has not been made, the simulation is consistent. Now, to make this query, $\mathcal{A}$ would have to solve CDH w.r.t. u_{p^*} and v_{i^*}. But if $\mathcal{A}$ ever does so, then $\mathcal{B}$ should have guessed differently and embedded u^* as u_{p^*} and v^* as v_{i^*} (assuming for the moment there are no Dec queries). $\mathcal{B}$'s guessing strategy will therefore be to guess the index i^* of the *first* key u_{i^*} generated during a query $\mathrm{Enc}(p^*)$ on the path to the challenge for which $\mathcal{A}$ will solve CDH via an RO query. (Note that $\mathcal{B}$ does not know the path; it simply guesses the index of an Enc query.) This is depicted in Fig. 3b.



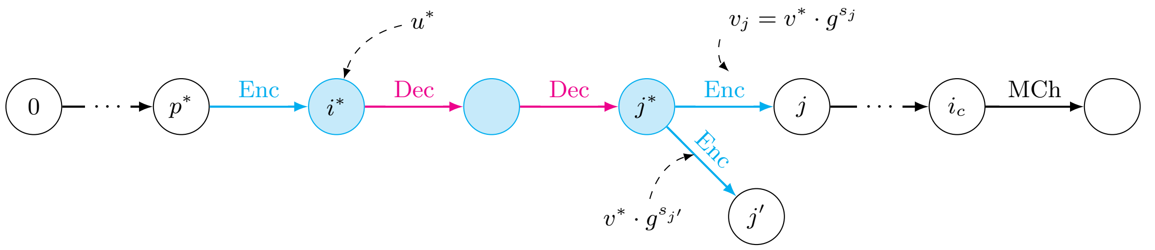
(a) Reduction $\mathcal{B}$ chose $x_0, \ldots, x_{p^*}$ itself and embedded u^* as u_{i^*} returned by a query Enc(p^*). Thus, $\mathcal{B}$ does not know the value d_{i^*} returned by the RO $H_1(u_{p^*}, w_{i^*})$ with $w_{i^*} = \mathrm{DH}(u_{p^*}, v_{i^*})$. But if $\mathcal{A}$ makes such a query, then $\mathcal{B}$ should have embedded u^* in p^*.



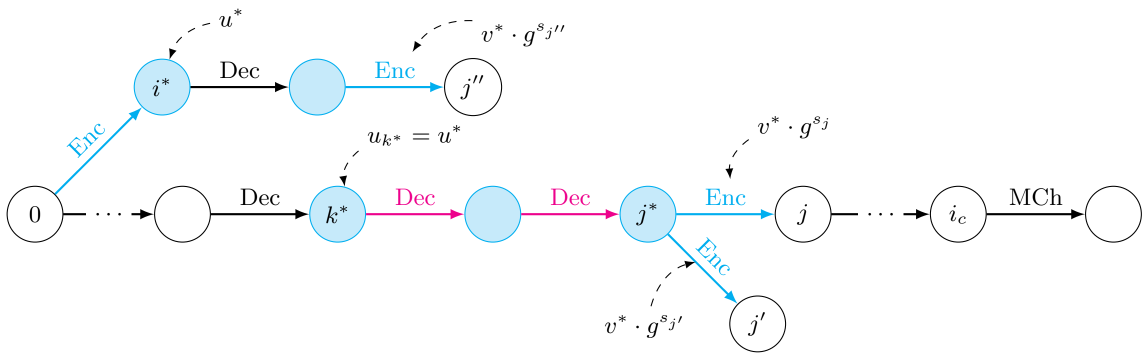
(b) Reduction $\mathcal{B}$ embeds u^* as the first key u_{i^*} on the path from 0 to i_c for which $\mathcal{A}$ will solve co-CDH via an RO query (u_{i^*}, w_j).



(c) The adversary can insert Dec-edges between i^* and j^*. $\mathcal{B}$ will extract d for such edges from the τ-proofs and use them to "translate" $w_j = \mathrm{DH}(u_{j^*}, v^*)$ to $w^* = \mathrm{DH}(u^*, v^*)$.



(d) If $\mathcal{A}$ creates an Enc-edge (j^*, j') then it is allowed to query Rev(j'). To answer such a query, $\mathcal{B}$ chooses the secret key $x_{j'}$ itself. Thus, $\mathcal{B}$ does not know $d_{j'}$ returned by the RO $H_1(u_{j^*}, w_{j'})$ with $w_{j'} = \mathrm{DH}(u_{j^*}, v_{j'})$. But if $\mathcal{A}$ makes such a query, then $\mathcal{B}$ can solve CDH if it embeds v^* in $v_{j'}$.



(e) Via a Dec query, $\mathcal{A}$ can also create a node k^* by copying the public key (and the tags) from some node i^* that is not on the path $0 \to i_c$. In this case $\mathcal{B}$ must embed u^* as the key in i^*. We are not able to answer Rev queries for descendants of i^* but they are not allowed as the descendants are in *chall-set*.

Fig. 3: Illustration of different executions of the member security experiment with reduction $\mathcal{B}$ to co-CDH running the adversary $\mathcal{A}$. Cyan marks nodes for which $\mathcal{B}$ does not know the secret key and edges for which $\mathcal{B}$ doesn't know the secret d. Magenta marks edges for which $\mathcal{B}$ will extract d from $\mathcal{A}$'s τ-proofs.

For now we only considered the case that $\mathcal{A}$ makes the query MChal(j^*) or Enc(j^*) assuming u_{j^*} was itself created during an Enc query; but u_{j^*} might have been created during a Dec query. That is, the attacked key (i.e., the one for which $\mathcal{A}$ solves CDH) has been generated by the adversary. Security now relies on the fact that ultimately the attacked key was derived (possibly via many Dec queries) from an honest key, say u_{i^*} (which might be u_0). This is depicted in Fig. 3c.

Since $\mathcal{A}$ must provide proofs τ_i for the hops from u_{i^*} to u_{j^*} (where τ_i proves knowledge of $d_i = x_i - x_{par_i}$), $\mathcal{B}$ can extract the values d_i and sum them to $d_{i^* \to j^*} := x_{j^*} - x_{i^*}$, which it can use to "translate" CDH solutions for u_{j^*} to u_{i^*}. Thus, it can embed u^* as u_{i^*} and embed $v^* = g^r$ as the ciphertext the adversary breaks. A solution $w = \mathrm{DH}(u_{j^*}, v^*)$ then yields a solution $w/(v^*)^{d_{i^* \to j^*}} = g^{x_{j^*} r}/g^{r(x_{j^*} - x_{i^*})} = \mathrm{DH}(u^*, v^*)$.

Our strategy is thus to guess the following index i^*: if the first attacked key is u_{j^*}, then i^* is the closest ancestor of j^* with a public key generated by the challenger. That is, at the latest $i^* = j^*$ (if j^* is generated during an Enc query), and at the earliest $i^* = 0$. Note that we do not need to guess j^* (where we embed v^*), as explained below.

Answering Rev *queries.* Say $\mathcal{B}$ embeds u^* as u_{i^*} and consider a query Enc(i^*), which creates a new key u_j. If node j turns out not to lie on the challenge path, then $\mathcal{A}$ is allowed to query Rev(j). However, if $\mathcal{B}$ ran Encaps to answer the query, setting $u_j := u^* \cdot g^{d_j}$ with $d_j := H_1(u_{i^*}, \mathrm{DH}(u_{i^*}, v_j))$, then it would not know $x_j = \log u_j$ to answer the Rev query.

But recall that $\mathcal{B}$ hopes that $\mathcal{A}$ attacks key u_{i^*}! Every time Enc or MChal is queried on i^*, the reduction thus embeds v^* from its co-CDH challenge into the ciphertext. In particular, using random self-reducibility, $\mathcal{B}$ chooses a uniform s_j and defines the new ciphertext as $v_j := v^* \cdot g^{s_j}$. If $\mathcal{A}$ ever queries $H_1(u_{i^*}, w_j)$ for $w_j := \mathrm{DH}(u_{i^*}, v_j)$, the game stops and $\mathcal{B}$ returns $w^* := w_j/(u^*)^{s_j} = g^{x^*(r+s_j)}/g^{x^* s_j} = \mathrm{DH}(u^*, v^*)$. On the other hand, as long as no such query is made, d_j is not defined, and thus $\mathcal{B}$ can simply sample x_j, set $u_j := g^{x_j}$ (which implicitly defines d_j) and simulate the proofs τ_j and π_j. This way, $\mathcal{B}$ can then answer the query Rev(j).

The case Enc(j^*) for an index j^* whose path from i^* consists of only Dec queries is dealt with similarly: $\mathcal{B}$ embeds $v^* \cdot g^{s_j}$ as v_j and samples x_j freshly. As long as the adversary does not query $H_1(u_i, w_j)$ with $w_j = \mathrm{DH}(u_{j^*}, v_j)$, the simulation is perfect. If the adversary makes that query, it can be translated back to a solution $\mathrm{DH}(u^*, v_j)$, and thus to $\mathrm{DH}(u^*, v^*)$, by extracting $d_{i^* \to j^*} = x_{j^*} - x^*$ from the τ-proofs provided by the adversary when making the Dec queries linking u_{i^*} to u_{j^*}: we have $w^* := w_j \cdot (u^*)^{-s_j} \cdot v_j^{-d_{i^* \to j^*}} = g^{(x^* + d_{i^* \to j^*})(r+s_j)} g^{-x^* s_j} g^{-(r+s_j) d_{i^* \to j^*}} = \mathrm{DH}(u^*, v^*)$. This is illustrated in Fig. 3d.

Extracting from adversarial proofs. Simulation-extractability of τ-proofs only lets us extract from proofs computed by the adversary (and not ones created by the simulator). So what happens if the adversary "copies" proofs simulated by the challenger?

In particular, consider the situation where we embedded our challenge key u^* as u_{i^*} and the adversary attacked one of its Dec-descendants u_{j^*}. If none of the key/proof pairs (u_i, τ_i) on the path from i^* to j^* appear elsewhere in the tree, then the statement/proof pairs are different from those of the simulated proofs, and we can extract their witnesses. On the other hand, assume that on this path, there is a pair (u_{k^*}, τ_{k^*}) which appears elsewhere as $(u_{k'}, \tau_{k'})$ in the tree. If (and only if) k' was created in a query Enc(i') and i' is a Dec-descendant of i^*, then $\tau_{k'}$ was simulated, and thus we cannot extract from $\tau_{k'} = \tau_{k^*}$. (Note that since for every u_k there is a unique valid π_k, we have $(u_{k^*}, \tau_{k^*}, \pi_{k^*}) = (u_{k'}, \tau_{k'}, \pi_{k'})$.)

However, this just means that we should have guessed differently: assume k^* is the last "copied" node on the path from i^* to j^*. If we had embedded our challenge key u^* as u_{k^*} (when we created it as $u_{k'}$ when answering an Enc query) then we could now solve

CDH: since, by assumption, no nodes between u_{k^*} and u_{j^*} are copied, we can extract from their τ-proofs and thus compute $d_{k^* \to j^*} = x_{j^*} - x_{k^*}$, which lets us shift a CDH solution for u_{j^*} to one for u_{k^*}. Note that we would not be able to answer Rev for k' and its Dec-descendants, but such queries are disallowed (as they are part of *chall-set*, cf. Fig. 1). This scenario is illustrated in Fig. 3e.

Our actual guess strategy is therefore: let u_{j^*} be the first key the adversary attacks during the game; then what is the index of the Enc query that creates the node (u_{k^*}, τ_{k^*}) so that when starting from u_{j^*} and moving up Dec-edges, (u_{k^*}, τ_{k^*}) is the first key/proof pair created by the challenger during an Enc query (at latest, this is u_0).

Answering Dec queries. We address answering decryption queries Dec(i) for nodes whose secret key is not known to the reduction. These are all nodes whose public key u^* is the embedded co-CDH instance, or any Dec-descendant of such nodes, marked in cyan in Fig. 3. Here we return to the ideas for proving CCA-security of DHIES, namely inspecting the random-oracle table. We moreover use the fact that CDH solutions can be checked via the pairing using the associated proof π_i: given a ciphertext v_j for key $u_i = g^{x_i}$, we have $K_j = H_2(u_i, w_j)$ with $w_j := \mathrm{DH}(u_i, v_j)$ and the latter can be efficiently checked: setting $\hat{u}_i := \pi_i \cdot h^{x_0} = h^{x_i}$ (where $h^{x_0} := \hat{u}^*$ if $i^* = 0$), check if $\mathrm{e}(w_j, h) \overset{?}{=} \mathrm{e}(v_j, \hat{u}_i)$.

So to decrypt ciphertext v_j for key u_i we do the following: if there has been a query $(u_i, \mathrm{DH}(u_i, v_j))$ to H_2, then we return the same key again; if there has not been such a query, we sample a fresh key K_j and (implicitly) program the random oracle: store an entry $(u_i, v_j, \perp, K_j)$, meaning that $(u_i, \mathrm{DH}(u_i, v_j))$ gets mapped to K_j. To detail how the Dec queries are answered, we first address programming of the random oracles.

Programming the random oracles. Answering Enc, Dec and MChal queries results in defining the entries of the random oracle tables for H_1 and H_2. The inputs are of the form (u, w), on which H_1 outputs d and H_2 which outputs K. For certain queries, these entries are partial, since the reduction does not know all inputs/outputs, meaning, the RO is programmed implicitly. The reduction thus stores RO entries of the form $(u, \hat{u}, v, w, u', d, K)$, some of whose components can be $\perp$. For $u = g^x$, the (non-$\perp$) values are: $\hat{u} = h^x$, $w = v^x = \mathrm{DH}(u, v)$, $u' = u \cdot g^d$ and d and K are the outputs of, respectively, H_1 and H_2, on input (u, w). Note that $\hat{u}$, w and u' are determined by the other values. During Enc and MChal queries, implicit programming happens at the following positions:

1. When embedding the key u^* as u_{i^*} for $i^* \neq 0$, letting $p^* := par_{i^*}$, the reduction implicitly defines the oracles at $(u_{p^*}, v_{i^*}^{x_{p^*}})$ (where x_{p^*} was chosen by the reduction); H_1 is set to $d_{i^*} := \log(u_{i^*}/u_{p^*})$ (unknown to the reduction) and H_2 is set to K_{i^*} (chosen by the reduction). When answering this query, the reduction thus stores the following entry (where $\hat{u}_{p^*} := h^{x_{p^*}}$):

$$(u_{p^*}, \hat{u}_{p^*}, v_{i^*}, \perp, u_{i^*}, \perp, K_{i^*})$$

(We put $\perp$ as the 4th component for consistency with the next case, although the reduction knows the value $v_{i^*}^{x_{p^*}}$.)
2. For any call of Enc or MChal at position i with $u_i = u^*$ or i being a Dec-descendant of a node with public key u^*, the reduction creates v_j (embedding v^* from its co-CDH instance) and u_j ($:= g^{x_j}$ for fresh j) and defines H_1 and H_2 at position $(u_i, \mathrm{DH}(u_i, v_j))$, which is unknown to the reduction. While the reduction chooses the value K_j at this position for H_2 (for the MChal query, K_j corresponds to "$K^{(1)}$"), it will not know the value $d_j = \log(u_j/u_i)$ for H_1. The reduction thus stores

$$(u_i, \hat{u}_i, v_j, \perp, u_j, \perp, K_j),$$

where, as above, $\hat{u}_i = \hat{u}^*$ if $i^* = 0$ and $\hat{u}_i := \pi_i \cdot h^{x_0} = h^{x_i}$ otherwise.

For every random-oracle query (u, w) the adversary makes, the reduction checks if $(u, w) = (u_i, \mathrm{DH}(u_i, v_j))$ holds when $i = i^*$, or $i = par_{i^*}$ or i is a Dec-descendant of i^*. It does this by checking $u \overset{?}{=} u_i$ and $\mathrm{e}(w, h) \overset{?}{=} \mathrm{e}(v_j, \hat{u}_i)$. (Note that such queries to H_1 cannot be answered, since the reduction does not know $d_j = \log(u_j/u_i)$.)

If this is the case for $i = par_{i^*}$, the reduction stops, since the guess i^* was wrong, as par_{i^*} would have been the right guess. If it happens for i^* or any of its Dec-descendants, the reduction stops and returns the co-CDH solution (computed as described above). Otherwise, fresh values d and K are sampled and a new entry $(u, \perp, \perp, w, u \cdot g^d, d, K)$ is created. We say that in this case the RO was *explicitly* programmed.

Details of answering Dec queries. Let us consider a query $\mathrm{Dec}(i', v', u', \tau', \pi')$. If τ' and π' are valid, a new (for now: half-)node is created. If i' is a full node, the oracle would do the following: run Decaps on $sk_{i'}$, that is, compute $d' := H_1(u_{i'}, \mathrm{DH}(u_{i'}, v'))$; check if $u' = u_{i'} \cdot g^{d'}$; if so, return $K := H_2(u_{i'}, \mathrm{DH}(u_{i'}, v'))$ and declare the new node a full node; else return $\perp$.

If i' is a half-node, then the reduction can simulate the Dec oracle perfectly, as the latter uses only public values. Moreover, if $i' \notin$ *chall-set*, then the reduction knows $sk_{i'}$ and can thus simulate the oracle perfectly as well.

For $i' \in$ *chall-set* and i' being a full node, the reduction uses its extended RO table as follows:

(i) If there is an entry $(u_{i'}, *, v', \perp, u'', d, K)$ for some u'', d (where possibly $d = \perp$) and K, then the RO was already implicitly programmed at $(u_{i'}, \mathrm{DH}(u_{i'}, v'))$ (either during and Enc or MChal query as described above, or during a Dec query as described below). The reduction checks if $u' = u''$ (as Decaps does) and if so, it declares the new node a full node and returns K; else, it declares the new node a half node and returns $\perp$.

(ii) Else if there is an entry $(u_{i'}, \perp, \perp, w, u'', d, K)$ for some u'' and $w = \mathrm{DH}(u_{i'}, v')$, which can be checked using $\hat{u}_{i'} = \pi_{i'} \cdot h^{x_0}$, then the RO was already explicitly programmed at $(u_{i'}, \mathrm{DH}(u_{i'}, v'))$. As above, the reduction checks if $u' = u''$ (as Decaps does); if so, it declares the new node a full node and returns K; else, it declares the new node a half node and returns $\perp$.

(iii) If none of the above apply, then sample d and K, create new entry

$$(u_{i'}, \hat{u}_{i'}, v', \perp, u_{i'} \cdot g^d, d, K)$$

and proceed as in Decaps. (Note that, with overwhelming probability, this will return $\perp$, since d will be inconsistent with $u_{i'}$ and u'.)

Finally, note that the only RO query that would reveal the challenge bit b is querying H_2 on $(u_{i_c}, \mathrm{DH}(u_{i_c}, v_{j_c}))$, where i_c is the value queried to MChal and j_c the current value of j when MChal was queried. "Explicit" queries are dealt with by our guessing strategy: if the guess i^* was correct then such a query is used to solve co-CDH. On the other hand, "implicit" queries via the Dec oracle cannot occur, since this would correspond to $\mathrm{Dec}(i', v', u', \tau', \pi')$ with $u' = u_{i_c}$ and $v' = v_{i_c}$, which is forbidden (as trivial wins).

Proof of Lemma 2. We define the set dec-set of an index i as all nodes that have the same public key u_i and member tag τ_i as i, together with all nodes created via Dec queries starting from these:

Definition 4 (Dec-set). *Let i be an index defined at some point during the experiment* $\mathrm{Exp}^{\textsf{CCA-M}}$*. Define the set* $\textsf{dec-set}(i) := \textsf{dec-closure}(\{i' \mid (u_{i'}, \tau_{i'}) = (u_i, \tau_i)\})$*, where* $\textsf{dec-closure}$ *is as in Fig. 1.*

Note that since $u_{i'} = u_i$ implies $\pi_{i'} = \pi_i$, we have $\textsf{dec-set}(i) := \textsf{dec-closure}(\{i' \mid u_{i'} = u_i \wedge \tau_{i'} = \tau_i \wedge \pi_{i'} = \pi_i\})$, meaning $\textsf{dec-set}(i)$ is the set $\textsf{chall-set}$ in Fig. 1 except with $\mathit{base} = \{i\}$.

The break event. Let $\mathcal{A}$ be an adversary playing in game $\mathrm{Exp}^{\textsf{CCA-M}}$. We define an event $\textsf{Brk}$, which intuitively occurs whenever $\mathcal{A}$ breaks CDH during the execution. That is, $\textsf{Brk}$ occurs if there is a node j^* with a child j created during a query $\mathrm{Enc}(j^*)$ or $\mathrm{MChal}(j^*)$, yielding ciphertext v_j, and $\mathcal{A}$ *breaks* the edge (j^*, j), i.e., it makes a random oracle query to H_1 or H_2 with input (u_{j^*}, w_j) with $w_j = \mathrm{DH}(u_{j^*}, v_j)$.

For this to be a *break*, we moreover require that the adversary cannot trivially compute w_j, e.g., by learning x_{j^*} via a Rev query. For this, we define the index $E(j^*)$ for any j^*. Intuitively, if $\mathcal{A}$ breaks an edge (j^*, j) for some j, then $E(j^*)$ is the index of the $\underline{\text{E}}$nc query creating a key of an ancestor of j in which the reduction to co-CDH should $\underline{\text{e}}$mbed u^*.

Definition 5 (Index E). *Let j be an index defined at some point during the security experiment* $\mathrm{Exp}^{\textsf{CCA-M}}$*. The index $E(j)$ is defined as follows. Let k be the closest ancestor of j s.t. (u_k, τ_k) was created by an Enc query ((u_k, τ_k) may have been "copied" to k from another node). If none exists, then $E(j) = 0$; else $E(j)$ is the first node created by an Enc query with values (u_k, τ_k).*

The event $\textsf{Brk}$ further requires that $\mathcal{A}$ does not corrupt any node in the set $S := \textsf{dec-set}(E(j^*))$. (Note that if $\mathcal{A}$ corrupts any such node, by moving along Dec edges ("upwards") and using the corresponding d values, it can compute x_{i^*} for $i^* = E(j^*)$, and then, again moving along Dec edges ("downwards"), compute x_{j^*}.)

Moreover, we also consider an edge "broken" if the adversary makes a random oracle query which breaks an edge that is only created *later*. (This only happens with negligible probability anyway, but defining breaks this way simplifies our analysis.)

If $\textsf{Brk}$ occurs then a reduction $\mathcal{B}$ can solve co-CDH if it embeds u^* as u_{i^*} with $i^* := E(j^*)$ (after correctly guessing i^*) and, for every Enc edge (j^*, j) with $j^* \in \textsf{dec-set}(i^*)$, it embeds v^* in v_j (using random self-reducibility). For this to work, we further need that $\mathcal{A}$ does not break the edge from the parent p^* of i^* to i^*. (The reduction could not answer the corresponding query to H_1, as it does not know its output $d_{i^*} := \log(u^*/u_{p^*})$.) Therefore, the event $\textsf{Brk}$ further requires that (j^*, j) is the *first* edge that was broken. This implies that (p^*, i^*) is not broken before (j^*, j), i.e., as long as the experiment lasts.

Definition 6 (Event $\textsf{Brk}$). *Let $\mathcal{A}$ be an adversary in game* $\mathrm{Exp}^{\textsf{CCA-M}}$*. We define an event* $\textsf{Brk-Edge}(j^*)$ *that occurs when both of the following hold:*

a) $\mathcal{A}$ queries $\mathrm{Enc}(j^*)$ or $\mathrm{MChal}(j^*)$, which returns the ciphertext v_j, and $\mathcal{A}$ queries H_1 or H_2 on input (u_{j^*}, w_j) with $w_j = \mathrm{DH}(u_{j^*}, v_j)$.
b) For all $j \in \textsf{dec-set}(E(j^*)) : \neg rev_j$.

$\textsf{Brk}(i^*)$ *occurs when $i^* = E(j^*)$, where j^* is the value for which* $\textsf{Brk-Edge}(j^*)$ *occurs first.*

(Note that in a) the random oracle query can also occur *before* the Enc or MChal query.)

We now show that the advantage of any adversary $\mathcal{A}$ in winning the game is upper-bounded by the probability of triggering $\textsf{Brk}(i^*)$ for some i^* in $\mathrm{Exp}^{\textsf{CCA-M}}$. The reason is that not triggering $\textsf{Brk}$ implies one of three things: Either $\mathcal{A}$ does not call MChal at all, or it queries MChal(i_c) for some i_c, but does not make the RO query breaking the MChal edge (i_c, j) (for $j^* = i_c$); then the challenge bit will be independent of its view. Or it makes a disallowed Rev query and loses anyway.

The proof then proceeds by guessing the index i^* and, when the guess was correct, bounding the probability of $\mathsf{Brk}(i^*)$ by the co-CDH advantage of a reduction $\mathcal{B}$, conditioned on correct simulation and extraction of member proofs.

Claim 0. $\mathsf{Adv}^{\mathsf{CCA\text{-}M}}(\mathcal{A}) \leq \Pr\left[\exists\, i^* : \mathsf{Brk}(i^*)\right].$

To show Claim 0, we let Win be the event that $\mathrm{Exp}^{\mathsf{CCA\text{-}M}}(\mathcal{A}) = 1$ and we define Brk (as above) as $\exists\, i^* : \mathsf{Brk}(i^*)$. We have

$$\Pr\left[\mathrm{Exp}^{\mathsf{CCA\text{-}M}}(\mathcal{A}) = 1\right] = \Pr\left[\mathsf{Win} \mid \mathsf{Brk}\right] \cdot \Pr[\mathsf{Brk}] + \underbrace{\Pr\left[\mathsf{Win} \mid \neg\mathsf{Brk}\right]}_{\leq 1/2\ (*)} \cdot (1 - \Pr[\mathsf{Brk}]), \qquad (1)$$

for which we show the bound $(*)$ below. Claim 0 then follows from (1) and $(*)$ since

$$\begin{aligned}\mathsf{Adv}^{\mathsf{CCA\text{-}M}}(\mathcal{A}) &= 2 \cdot \Pr\left[\mathrm{Exp}^{\mathsf{CCA\text{-}M}}(\mathcal{A}) = 1\right] - 1 \\ &\leq 2 \cdot \Pr\left[\mathsf{Win} \mid \mathsf{Brk}\right] \cdot \Pr[\mathsf{Brk}] - \Pr[\mathsf{Brk}] \leq \Pr[\mathsf{Brk}].\end{aligned}$$

To show $(*)$ $\Pr\left[\mathsf{Win} \mid \neg\mathsf{Brk}\right] \leq \frac{1}{2}$, we partition the event $\neg\mathsf{Brk}$:

Case 1: $\mathcal{A}$ makes no query MChal. Then the bit b is information-theoretically hidden from $\mathcal{A}$ and the probability of Win is at most 1/2.

Case 2: $\mathcal{A}$ makes a query MChal(i_c) for some i_c and an "illegal" query to oracle Rev, that is, at the end of game $\mathrm{Exp}^{\mathsf{CCA\text{-}M}}$, for some $j \in \mathsf{chall\text{-}set}(i_c) : rev_j$, with $\mathsf{chall\text{-}set}$ as defined in Fig. 1. Then, by definition of the $\mathrm{Exp}^{\mathsf{CCA\text{-}M}}$, the probability of Win is 0.

Case 3: None of the above, that is, $\mathcal{A}$ makes a query MChal(i_c) and

$$\text{for all } j \in \mathsf{chall\text{-}set}(i_c) : \neg rev_j. \qquad (2)$$

We argue that in Case 3, the probability of Win (conditioned on $\neg\mathsf{Brk}$) is also bounded by 1/2. We start with showing the following:

$(**)$ If $\neg\mathsf{Brk}$ and Case 3 happen then $\mathcal{A}$ does not query H_2 on input (u_{i_c}, w_{j_c}) with $w_{j_c} = \mathrm{DH}(u_{i_c}, v_{j_c})$ for v_{j_c} returned by MChal(i_c).

Indeed, assume towards a contradiction that $\mathcal{A}$ made this query to H_2. Then condition a) of Brk is satisfied for $j^* := i_c$. Moreover, let $i^* := E(i_c)$. We show that $(***)$ $\mathsf{dec\text{-}set}(i^*) \subseteq \mathsf{chall\text{-}set}(i_c)$: Indeed, if k is as in Definition 5 then $\mathsf{dec\text{-}set}(i^*) = \mathsf{dec\text{-}set}(k)$ and k is an ancestor of i_c. The latter means that $k \in base$ in Fig. 1, so $\mathsf{dec\text{-}set}(k) \subseteq \mathsf{chall\text{-}set}(i_c)$ (since $\mathsf{dec\text{-}set}$ and $\mathsf{chall\text{-}set}$ are computed the same way).

Eq. (2) together with $(***)$ implies that condition b) of Brk is satisfied for j^*. Thus, if no other value makes Brk occur earlier then j^* is the first value and thus satisfies a) and b), which means that Brk occurs and contradicts our assumptions. This shows $(**)$.

Now, since (as just shown) $\mathcal{A}$ does not query $H_2(u_{i_c}, \mathrm{DH}(u_{i_c}, v_{j_c}))$, which is what defines $K^{(1)}$, the challenge keys $K^{(1)}$ and $K^{(0)}$ are both independently random. The adversary's view is thus independent of the bit b, which concludes showing $(*)$ and thus the proof of Claim 0.

Sequence of hybrids. For convenience, we next define a sequence of hybrid experiments, starting with $\mathrm{Exp}^{\mathsf{CCA\text{-}M}}$, each with its version of the break event.

Hybrid 0. This is the experiment $\mathrm{Exp}_0^{\mathsf{CCA\text{-}M}} = \mathrm{Exp}^{\mathsf{CCA\text{-}M}}$.

Hybrid 1 *(Guessing the attacked key).* The experiment $\mathrm{Exp}_1^{\mathsf{CCA\text{-}M}}$ differs from Hybrid 0 in that at the beginning the challenger guesses the index i^* that makes the event Brk_0 occur. The experiment ends as soon as either $\mathsf{Brk}_0(i^*)$ occurs, or the guess becomes incorrect, i.e., $\mathsf{Brk}_0(i^*)$ cannot occur. We define Brk_1^* as the event that in Hybrid 1 $\mathsf{Brk}_0(i^*)$ occurs for the i^* guessed by the challenger.

Claim 1. $\Pr[\exists i^* : \mathsf{Brk}_0(i^*)] \leq (n_e + 1) \cdot \Pr[\mathsf{Brk}_1^*]$, where n_e is an upper bound on the number of $\mathcal{A}$'s Enc queries.

The claim is straightforward given that there are $n_e + 1$ possible guesses for i^*: n_e possible indices created by Enc queries and node 0.

Hybrid 2 *(Simulating member and joiner tags).* In Hybrid 2, the event Brk_2^* is defined as in Hybrid 1 and the experiment $\mathrm{Exp}_2^{\mathsf{CCA\text{-}M}}$ differs from $\mathrm{Exp}_1^{\mathsf{CCA\text{-}M}}$ as follows:

- If $i^* > 0$ then in the response of the query $\mathrm{Enc}(p^*)$ creating node i^*, the proofs τ_{i^*} and π_{i^*} are simulated: $\tau_{i^*} \leftarrow \mathsf{PoL.Simulate}_{H_3}((u_{p^*}, u_{i^*}))$ and $\pi_{i^*} = \hat{u}_{i^*}/h^{x_0}$. (Jumping ahead, the reduction to co-CDH will have chosen x_0 itself and use $\hat{u}_{i^*} = \hat{u}^*$ from its co-CDH instance.)
- In the responses of all queries $\mathrm{Enc}(i)$ and $\mathrm{MChal}(i)$ queries for $i \in \mathsf{dec\text{-}set}(i^*)$, the proofs τ_j and π_j are also simulated: $\tau_j \leftarrow \mathsf{PoL.Simulate}_{H_3}((u_i, u_j))$ and π_j computed as follows:
  - If $i^* = 0$, then $\pi_j \leftarrow h^{x_j}/\hat{u}_0$, where $\hat{u}_0 = h^{x_0}$ (jumping ahead, the reduction to co-CDH will use $\hat{u}_0 = \hat{u}^*$ from its co-CDH instance).
  - If $i^* > 0$, then $\pi_j \leftarrow h^{x_j - x_0}$ (jumping ahead, the reduction knows x_j and x_0).

Note that in Hybrid 2, for all $i \in \{par_{i^*}\} \cup \mathsf{dec\text{-}set}(i^*)$ and nodes j created via $\mathrm{Enc}(i)$ or $\mathrm{MChal}(i)$, no values x_i and r_j (i.e., the logarithm of v_j) are used to compute proofs τ_j and π_j.

Claim 2. $\Pr[\mathsf{Brk}_1^*] \leq \epsilon^{\mathrm{sim}}_{\mathsf{PoL},n_e+1} + \Pr[\mathsf{Brk}_2^*]$, where n_e is an upper bound on $\mathcal{A}$'s Enc queries and $\epsilon^{\mathrm{sim}}_{\mathsf{PoL},n}$ is the simulation error of PoL when simulating n proofs.

The claim is straightforward, noting that at most $n_e + 1$ τ-proofs are simulated: n_e for Enc queries and one for the MChal query. Moreover, the simulation of the proofs π_j is perfect.

Hybrid 3 *(Decrypting keys K).* In Hybrid 3, the event Brk_3^* is defined as in Hybrid 2 and the experiment $\mathrm{Exp}_3^{\mathsf{CCA\text{-}M}}$ differs from $\mathrm{Exp}_2^{\mathsf{CCA\text{-}M}}$ in how it defines entries in the random oracle tables and answers Dec queries for indices in $\mathsf{dec\text{-}set}(i^*)$, following the strategy outlaid in the proof intuition (p. 24).

Recall that the UPKE scheme always calls H_1 and H_2 together on the same inputs $(u, w) = (g^x, v^x)$. Thus for convenience, we store outputs of the two random oracles together: the challenger keeps a list of entries $(g^x, h^x, v, v^x, g^{x+d}, d, K)$, denoting that on input (g^x, v^x), the oracle H_1 outputs d and H_2 outputs K. Several components of an entry might be set to $\perp$ (because they will be unknown to the final reduction).

Enc queries related to challenge. When answering queries $\mathrm{Enc}(i)$ and $\mathrm{MChal}(i)$ for $i \in \{par_{i^*}\} \cup \mathsf{dec\text{-}set}(i^*)$, the challenger stores entries $(u_i, \hat{u}_i, v_j, \perp, u_j, \perp, K)$, where j is the current value of the counter. The 4th component $w_j := \mathrm{DH}(u_i, v_j)$ is implicitly defined by u_i and v_j; the 6th component d is implicitly defined by u_i and u_j as d s.t. $u_j = u_i \cdot g^d$.

If however, there already exists an entry $(u_i, \perp, \perp, w_j, *, *, *)$ with $w_j := \mathrm{DH}(u_i, v_j)$ (resulting from an RO query by $\mathcal{A}$), then this Enc query triggered Brk_2^* (and Brk_3^*), so the experiment stops (and the final reduction will solve its co-CDH instance).

Random oracle queries. For every query (u, w) to H_1 or H_2:

1. Check if the query was implicitly programmed: search for an entry $(u_i, \hat{u}_i, v_j, \perp, u_j, d_j, K_j)$ with $u_i = u$ and $\mathrm{DH}(u_i, v_j) = w$ by checking if $\mathrm{e}(w, h) \overset{?}{=} \mathrm{e}(v_j, \hat{u}_i)$. If such an entry exists:

   (i) If $d_j \neq \perp$, i.e., the entry was created during a Dec query (see below), then return d_j for an H_1 query or K_j (which is never $\perp$) for an H_2 query.
   (ii) If $d_j = \perp$ then the entry was created during an Enc query (see above). Therefore, we must have $i = par_{i^*}$ or $i \in \mathsf{dec\text{-}set}(i^*)$. In the first case, the current RO query means that the guess i^* is not correct so Brk_2^* (and Brk_3^*) does not occur and the experiment stops. In the latter case, this RO query precisely triggers Brk_2^* (and Brk_3^*), so the experiment stops (and the final reduction will solve its co-CDH instance).

2. Check if the query was explicitly programmed: if an entry $(u, \perp, \perp, w, u', d, K)$ exists then return d for an H_1 query or K for an H_2 query.
3. Else, sample fresh d and K and store the entry $(u, \perp, \perp, w, u \cdot g^d, d, K)$ (thus programming the RO explicitly).

Dec queries. When answering $\mathrm{Dec}(i', v', u', \tau', \pi')$ for $i' \in \mathsf{dec\text{-}set}(i^*)$, return $\perp$ if τ' or π' is invalid. Otherwise let $\hat{u}_{i'} := \hat{u}^*$ if $i^* = 0$ and else $\hat{u}_{i'} := \pi_{i'} \cdot h^{x_0}$ and process the query as follows:

(i) If there is an entry $(u_{i'}, \hat{u}_{i'}, v', \perp, u'', d, K)$ (where d could be $\perp$) then:
– check if $u' = u''$ (as Decaps does);
– if so and i' is a full node, then declare the new node a full node and return K;
– else, declare the new node a half node and return $\perp$.
(ii) Else if there is an entry $(u_{i'}, \perp, \perp, w, u'', d, K)$ for w satisfying $\mathrm{e}(w, h) = \mathrm{e}(v, \hat{u}_{i'})$, then:
– if $u' = u''$ and i' is a full node, declare the new node a full node and return K;
– else declare the new node a half node and return $\perp$.
(iii) If none of the above apply, sample fresh d and K and create entry $(u_{i'}, \hat{u}_{i'}, v', \perp, u_{i'} \cdot g^d, d, K)$ (thus implicitly defining the 4th component as $\mathrm{DH}(u_{i'}, v')$). Then proceed "honestly" as in the previous hybrids.

Note that Hybrid 3 can be efficiently simulated without knowing x_{i^*} s.t. $u_{i^*} = g^{x_{i^*}}$ nor the values r_j for nodes j created via Enc/MChal(i) for $i \in \mathsf{dec\text{-}set}(i^*)$.

Claim 3. $\Pr[\mathsf{Brk}_2^* = 1] = \Pr[\mathsf{Brk}_3^* = 1]$.

The claim follows, since every "implicit" RO table entry (i.e., one created during an Enc query or in Step (iii) of a Dec query) could be translated to an "explicit" entry, i.e., one created in Step 3 of a RO query. The experiment defines all entries consistently (every position (u, w) is uniquely defined) so the game proceeds as if it only used "explicit" entries, as it does in Hybrid 2. Note that when the experiment stops, it is already determined if Brk_2^* / Brk_3^* occurred.

Hybrid 4 *(Extraction).* The experiment $\mathrm{Exp}_4^{\mathsf{CCA\text{-}M}}$ differs from $\mathrm{Exp}_3^{\mathsf{CCA\text{-}M}}$ as follows: When Brk^* occurs, let $i_0, i_1, \ldots, i_\ell \in \mathsf{dec\text{-}set}(i^*)$ denote the path with $(u_{i_0}, \tau_{i_0}) = (u_{i^*}, \tau_{i^*})$ and $i_\ell = j^*$, with j^* as in Definition 6 (i.e., j^* is the "attacked" node). The challenger extracts witnesses $d_1, \ldots, d_\ell$ from the proofs $\tau_{i_1}, \ldots, \tau_{i_\ell}$ sent by $\mathcal{A}$ to the Dec oracle when nodes $i_1, \ldots, i_\ell$ were created. The event Brk_4^* occurs if and only if Brk_3^* occurs and extraction succeeds (which can be checked efficiently).

Claim 4. There exists an adversary $\mathcal{B}'$ against strong simulation-extractability of PoL s.t.

$$\Pr\left[\mathsf{Brk}_3^*\right] \leq \epsilon^{\text{ext}}_{\mathsf{PoL},n_d}(\mathcal{B}') + \Pr\left[\mathsf{Brk}_4^*\right],$$

where n_d is an upper bound on $\mathcal{A}$'s Dec queries and $\epsilon^{\text{ext}}_{\mathsf{PoL},n}(\mathcal{B}')$ is $\mathcal{B}'s$ advantage for n proofs.

The claim follows from strong simulation-extractability (sSE) of PoL: define an adversary $\mathcal{B}'$ that simulates $\text{Exp}_4^{\mathsf{CCA\text{-}M}}$ for $\mathcal{A}$, except that instead of simulating the proofs τ itself (see Hybrid 2), it queries them to its simulation oracle. Further, when Brk_3^* occurs then the experiment stops and instead of extracting from the proofs $\tau_{i_1}, \ldots, \tau_{i_\ell}$ as the challenger would, $\mathcal{B}'$ outputs these proofs and the corresponding statements $(u_{i_0}, u_{i_1}), \ldots, (u_{i_{\ell-1}}, u_{i_\ell})$.

Observe that $\mathcal{B}'$ emulates $\text{Exp}_4^{\mathsf{CCA\text{-}M}}$ perfectly until Brk_3 occurs. Further, Hybrids 3 and 4 are identical unless extraction does not succeed for all τ_{i_j}. We show that in this case $\mathcal{B}'$ breaks sSE. This is the case iff (1) the extractor failed on a valid statement/proof pair outputted by $\mathcal{B}'$ and (2) none of these pairs correspond to a simulation oracle query/response.

For (1) note that all proofs are valid, as otherwise the corresponding node would not have been created. To show that (2) holds as well, assume towards a contradiction that Brk_3^* occurs, but for some $i' \in \{i_1, \ldots, i_\ell\}$ the statement $(u_{par_{i'}}, u_{i'})$ was queried to the simulation oracle and answered with $\tau_{i'}$; further, let i' be the largest index for which this is the case. This means that there is a node k^* created during the query $\text{Enc}(par_{k^*})$ with $(u_{k^*}, \tau_{k^*}) = (u_{i'}, \tau_{i'})$ and $par_{k^*} \in \{par_{i^*}\} \cup \mathsf{dec\text{-}set}(i^*)$ (which is the set of nodes for which Enc simulates proofs). But this means that i', and not i_0, is the closest ancestor of j^* whose values were created by an Enc query (namely the query generating node k^*). Thus, k^*, and not i^*, is the correct guess and thus Brk_3^* did not occur, which is a contradiction. (Also note that $(u_{i'}, \tau_{i'}) \neq (u_{i_0}, \tau_{i_0})$, as otherwise, the path $i_0, \ldots, i', \ldots, i_\ell$ would start at i'. Therefore, also the values of k^* and i^* are different and thus $k^* \neq i^*$.)

Final reduction *(to co-CDH).* The last step in the proof is to break co-CDH whenever Brk_4^* occurs.

Claim 5. For any adversary $\mathcal{A}$, there exists a reduction $\mathcal{B}$ s.t.

$$\Pr\left[\mathsf{Brk}_4^*\right] \leq \mathsf{Adv}_{\mathcal{G}}^{\mathsf{co\text{-}CDH}}(\mathcal{B}).$$

To prove the claim, we construct a reduction $\mathcal{B}$, which is given a co-CDH instance $(u^*, \hat{u}^*, v^*)$ and simulates $\text{Exp}_4^{\mathsf{CCA\text{-}M}}$ for $\mathcal{A}$ as follows:

1. $\mathcal{B}$ embeds u^* as u_{i^*}, computes $\pi_{i^*} := \hat{u}^*/h^{x_0}$ (with $\pi_{i^*} = h^0$ if $i^* = 0$) and simulates τ_{i^*} if $i^* \neq 0$.
2. For each query $\text{Enc}(i)$ or $\text{MChal}(i)$ for $i \in \mathsf{dec\text{-}set}(i^*)$ creating node j:
   a) $\mathcal{B}$ samples $s_j \leftarrow_\$ \mathbb{Z}_p$ and embeds v^* in the ciphertext $v_j := v^* \cdot g^{s_j}$. (Observe that v_j is distributed identically as in $\text{Exp}_4^{\mathsf{CCA\text{-}M}}$ where it is chosen as g^{r_j} for $r_j \leftarrow_\$ \mathbb{Z}_p$.)
   b) $\mathcal{B}$ samples $x_j \leftarrow_\$ \mathbb{Z}_p$ and sets $u_j := g^{x_j}$. It simulates the proofs τ_j and π_j as of Hybrid 2, so $\mathcal{B}$ does not need to know d_j. (Note that $\mathcal{B}$ thus knows the secret key x_i of each node $i \notin \mathsf{dec\text{-}set}(i^*)$.)
   c) $\mathcal{B}$ generates the key K_j independently at random.
3. $\mathcal{B}$ simulates Enc, Rev and Dec queries for $i \notin \mathsf{dec\text{-}set}(i^*)$ perfectly using the secret keys it generated.

4. $\mathcal{B}$ answers random-oracle queries and queries $\text{Dec}(i', v', u', \tau', \pi')$ for $i \in \textsf{dec-set}(i^*)$ without knowing $x_{i'}$ as defined in Hybrid 3.
5. Whenever $\mathcal{A}$ queries H_1 or H_2 on some (u, w), $\mathcal{B}$ follows Steps 1 and 2 in Hybrid 3. When in 1.(ii) the simulation stops, i.e., there is an entry $(u, *, v_j, \bot, *, \bot, *)$ with $w = \text{DH}(u, v_j)$, then $\mathcal{B}$ computes the co-CDH solution as follows. Assuming the guess i^* is correct, we have $v_j = v^* \cdot g^{s_j}$. Moreover there is a path $i_0, \dots, i_\ell$ with $u_{i_0} = u^*$ and $i_\ell = j^*$ with $u = u_{j^*}$. Let $d_1, \dots, d_\ell$ be the witnesses for $\tau_{i_1}, \dots, \tau_{i_\ell}$ extracted as of Hybrid 4 and set $d_{i^* \to j^*} := d_1 + \cdots + d_\ell$ (if $i^* = i^*$ then $d_{i^* \to j^*} = 0$). Return $w^* := w / ((u^*)^{s_j} \cdot v_j^{d_{i^* \to j^*}})$.

We first show that $\mathcal{B}$ simulates $\text{Exp}_4^{\textsf{CCA-M}}$ perfectly until $\textsf{Brk}_4^*$ occurs. To this end, observe first that u_j and K_j generated in Step 2 are distributed identically as in $\text{Exp}_4^{\textsf{CCA-M}}$ unless $\mathcal{A}$ queries some $(u_i, w_j := \text{DH}(u_i, v_j))$ to the RO. However, as soon as this happens, $\mathcal{B}$ stops the experiment in Step 5. Next, observe that the key u_{i^*} embedded in Step 1 is distributed identically as in $\text{Exp}_4^{\textsf{CCA-M}}$ unless $i^* > 0$ and $\mathcal{A}$ makes a query $(u_{p^*}, w_{i^*} := \text{DH}(u_{p^*}, v_{i^*}))$ to H_1, where p^* is the parent of i^* (since $u_{i^*} = u_i \cdot g^{H_1(u_{p^*}, w_{i^*})}$). However, if $\mathcal{A}$ makes such a query, then i^* is not the correct guess, contradicting $\textsf{Brk}_4^*$. Indeed, the oldest twin of the closest ancestor of p^* created by an Enc query (or $i = 0$) would be the correct guess, since p^* satisfies a) and b) in Definition 6 first.

Therefore, the only situation in which $\text{Exp}_4^{\textsf{CCA-M}}$ and the simulation can differ is when $\mathcal{A}$ makes a $\text{Rev}(i)$ query that is allowed in $\text{Exp}_4^{\textsf{CCA-M}}$ but which $\mathcal{B}$ cannot answer. This cannot occur, because the only values for which $\mathcal{B}$ does not know the secret key are all $i \in \textsf{dec-set}(i^*)$. Since, as argued in $(***)$ in the proof of Claim 0, we have $\textsf{dec-set}(i^*) \subseteq \textsf{chall-set}(i_c)$, no such corruptions are not allowed.

It remains to argue that $\mathcal{B}$ finds the correct solution w^*. To this end, observe that by correctness of extraction, we have $u_{j^*} = u^* \cdot g^{d_{i^* \to j^*}}$. Therefore, setting $u^* = g^{x^*}$ and $v^* = g^{r^*}$, we get

$$\begin{aligned} w = \text{DH}(u_{j^*}, v_j) &= \text{DH}(u^* \cdot g^{d_{i^* \to j^*}}, v^* \cdot g^{s_j}) = g^{(x^* + d_{i^* \to j^*})(r^* + s_j)} \\ &= g^{x^* r^*} \cdot g^{x^* s_j} \cdot g^{d_{i^* \to j^*}(r^* + s_j)} = \text{DH}(u^*, v^*) \cdot (u^*)^{s_j} \cdot v_j^{d_{i^* \to j^*}}. \end{aligned}$$

The lemma now follows by combining Claims 0 to 5. □

5.2 Joiner Security

We next prove the following lemma, which formalizes the joiner security, $\textsf{CCA-J}$, of our UKEM scheme.

Lemma 3. *Let $\mathcal{G}$ be an asymmetric bilinear group. If* $\textsf{PoL}$ *is a simulation-extractable proof system, co-CDH holds for $\mathcal{G}$ and adversary $\mathcal{A}$ is algebraic in $\hat{\mathbb{G}}$, the UKEM construction from Figure 2 is* $\textsf{CCA-J}$ *secure in ROM. More precisely, for any algebraic adversary $\mathcal{A}$, there exist reductions $\mathcal{B}$ and $\mathcal{B}'$ such that*

$$\textsf{Adv}^{\textsf{CCA-J}}(\mathcal{A}) \leq \epsilon^{\text{sim}}_{\textsf{PoL}, n_e} + \epsilon^{\text{ext}}_{\textsf{PoL}, n_d}(\mathcal{B}') + \textsf{Adv}^{\text{co-CDH}}_{\mathcal{G}}(\mathcal{B}),$$

where n_e and n_d are upper bounds on the number of $\mathcal{A}$'s Enc and Dec queries, respectively.

Proof intuition. We build upon ideas the proof of Lemma 2 (member security), but for joiner security the argument simplifies and there is no guessing of nodes. The difference between the experiments is that instead of MChal, the adversary $\mathcal{A}$ now calls $\text{JChal}(u', \pi')$.

Accordingly, the reduction $\mathcal{B}$ against co-CDH now embeds v^* in the ciphertext v' returned by JChal; specifically, using random self-reducibility, it sets $v' := v^* \cdot g^{s'}$ for a random s'. The security of v' encrypted to u' hinges on the link between u' and the honest u_0 via the associated proof π'. Thus, the value u^* of the co-CDH challenge is embedded as u_0.

More precisely, unless $\mathcal{A}$ queries H_2 on $w' := \mathrm{DH}(u', v')$, both the "random" key $K^{(0)}$ and the "real" key $K^{(1)} = H_2(u', w')$ are random and independent, so $\mathcal{A}$'s advantage is 0. On the other hand, if $\mathcal{A}$ makes such an RO query, $\mathcal{B}$ can compute the co-CDH solution by extracting from the proof π' to move the solution from u' to $u_0 = u^*$.

Extracting from the proof π'. Since $\mathcal{A}$ is algebraic, when it calls JChal(u', π'), $\mathcal{B}$ can extract the representation of π' as a linear combination of all $\hat{\mathbb{G}}$ elements given to $\mathcal{A}$ so far, which are (precisely) the π_j proofs returned by the Enc oracle. $\mathcal{B}$ knows the logarithm of each such π_j because it emulates the Enc oracle by running Encaps honestly (while $\mathcal{B}$ does not know any secret keys). Thus, $\mathcal{B}$ can use the representation of π' and the known logarithms to compute the logarithm d' of π'. Since π' is valid, d' is equal to the logarithm of $u'/u_0 = u'/u^*$. Thus $\mathcal{B}$ can use d' and s' chosen when embedding v' as $v' = v^* \cdot g^{s'}$ to translate $w' = \mathrm{DH}(u', v') = \mathrm{DH}(u^* \cdot g^{d'}, v^* \cdot g^{s'})$ from $\mathcal{A}$'s RO query to the solution $w^* = \mathrm{DH}(u^*, v^*)$ analogously to the reduction for member security: $w^* = w' \cdot (u^*)^{-s'} \cdot (v')^{-d'}$.

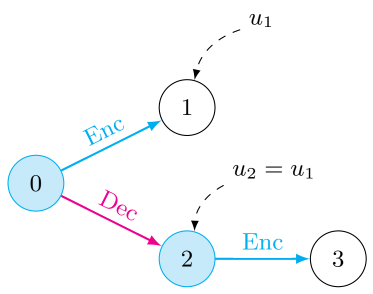
Answering Rev queries. If, after the JChal call, $\mathcal{A}$ makes a query Enc(i) creating a node j, then it is allowed to query Rev(j). $\mathcal{B}$ deals with such queries the same way as the reduction for member security: It samples the secret key x_j itself. This implies that $\mathcal{B}$ cannot answer a query $(u_i, \mathrm{DH}(u_i, v_j))$ to H_1, which should return $d_j = \log(g^{x_j}/u_i)$. But again, such a query would allow $\mathcal{B}$ to solve its co-CDH instance, had it embedded v^* in v_j.

In more detail, recall that Rev queries are allowed for nodes outside of *chall-set* which is the *dec-closure* of all nodes (and their duplicates) created before the JChal call. If after the JChal query $\mathcal{A}$ queries Enc(i) for some $i \in$ *chall-set*, $\mathcal{B}$ samples x_j itself and returns $v_j = v^* \cdot g^{s_j}$ for a random s_j together with simulated proofs π_j and τ_j. If $\mathcal{A}$ later "breaks" v_j by making an RO query (u_i, w_j) with $w_j = \mathrm{DH}(u_i, v_j)$ then $\mathcal{B}$ translates w_j to the co-CDH solution $\mathrm{DH}(u^*, v^*)$: it does so by collecting and summing all d values on the path from node 0 to node i: for any Dec edge, $\mathcal{B}$ extracts d from the τ proof provided by $\mathcal{A}$; for any Enc edge, $\mathcal{B}$ generated d itself, as all Enc edges in *chall-set* were created before JChal and hence by running Encaps. Knowing $d_{0\to i}$ s.t. $u_i = u_0 \cdot g^{d_{0\to i}} = u^* \cdot g^{d_{0\to i}}$ and s_j s.t. $v_j = v^* \cdot g^{s_j}$, the reduction can compute $\mathrm{DH}(u^*, v^*) = \mathrm{DH}(u_i, v_j) \cdot (u^*)^{-s_j} \cdot v_j^{-d_{0\to i}}$.

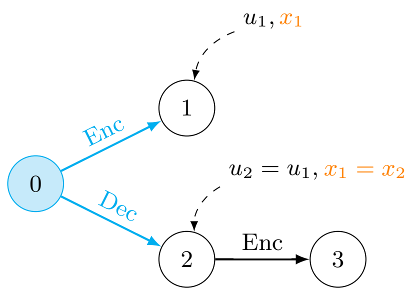
Observe that for the above "simulated" edges, $\mathcal{B}$ does not know the logarithm of the simulated π_j. This does not affect extraction from the proof π' above, since extraction is done when JChal is called, thus before any proofs are simulated.

Extracting from adversarial τ proofs. Say $\mathcal{A}$ breaks some v_j embedded in the response to Enc(i) as described above. Simulation-extractability allows $\mathcal{B}$ to extract from τ-proofs for Dec edges as long as these were not simulated. However, $\mathcal{A}$ could copy a node with a simulated proof via a Dec query. Therefore, we need to modify $\mathcal{B}$'s strategy, analogously to the proof of member security.

Consider the following example illustrated in Fig. 4: $\mathcal{A}$ starts by calling JChal (not depicted) and then queries Enc(0) creating node 1, at which point $\mathcal{B}$ picks x_1 and sets $u_1 = g^{x_1}$ as described above. Now $\mathcal{A}$ forwards the outputs of the above Enc query to Dec(0), creating node 2 with $(u_2, \tau_2) = (u_1, \tau_1)$. Finally, $\mathcal{A}$ queries Enc(2), which creates node 3. Since node 2 is in *chall-set*, $\mathcal{B}$, following the above strategy, would choose a fresh key x_3 for $u_3 := g^{x_3}$. However, if $\mathcal{A}$ queries $(u_2, \mathrm{DH}(u_2, v_3))$ to H_1, then $\mathcal{B}$ would not be



(a) Bad strategy: Picking x_3 itself and embedding a challenge in v_3 would not work, because $\mathcal{B}$ would not know d_3 to answer the RO query on $(u_2, \mathrm{DH}(u_2, v_3))$



(b) Good strategy: $\mathcal{B}$ knows that $x_2 = x_1$, which it generated itself, so it can answer the Enc query using Encaps, just like outside *chall-set*.

Fig. 4: Illustration of why $\mathcal{B}$'s first strategy does not work. Cyan marks nodes for which the reduction does not know the secret key and edges for which the reduction doesn't know the secret d. Magenta marks edges for which the reduction will extract d from the adversary's τ-proofs.

able to answer, since it does not know $d_3 := x_3 - x_2$; moreover, the value $\mathrm{DH}(u_2, v_3)$ is of no use, as $\mathcal{B}$ can compute it itself as $v_3^{x_2} = v_3^{x_1}$. But in this situation, $\mathcal{B}$ should have just computed u_3 honestly as $u_3 = u_2 \cdot g^{d_3}$. It could then still answer Rev(3), as required, since it knows $x_3 = x_1 + d_3$.

$\mathcal{B}$'s strategy is thus the following: u^* is embedded as u_0, and before the JChal query, every Enc query is answered by running Encaps (and thus $\mathcal{B}$ does not know the resulting secret key). After the JChal query, every query Enc(i) creating node j must be answered in a way so $\mathcal{B}$ knows the resulting secret key x_j. We distinguish two cases: (1) $\mathcal{B}$ knows x_i, or x_k for any Dec-"ancestor" k of i: then $\mathcal{B}$, knowing x_i, runs Encaps, and will thus know x_j. (2) Else $\mathcal{B}$ sets the resulting key as $u_j := g^{x_j}$ and $v_j := v^* \cdot g^{s_j}$ for fresh x_j, s_j, and simulates the proofs.

Note that for every $i \notin$ *chall-set*, $\mathcal{B}$ either knows x_i, or it can compute it by running Decaps between the node k for which it knows x_k and i in order to derive x_i. Therefore, $\mathcal{B}$ can answer all Rev queries for such i.

Moreover, when $\mathcal{A}$ makes an unanswerable RO query, $\mathcal{B}$ can use it to break co-CDH. Any such query is of the form $(u_i, w_j = \mathrm{DH}(u_i, v_j))$ where j is a node created in mode (2) above (for which $\mathcal{B}$ does not know d_j). $\mathcal{B}$ extracts all d values from the proofs τ on the path from the root to node i. This must succeed as long as none of the proofs was simulated, which we show next.

Towards a contradiction, assume that for some k on that path, τ_k for the statement (u_{par_k}, u_k) was simulated. $\mathcal{B}$ must have simulated the proof when, after the JChal call, $\mathcal{A}$ called Enc($par_{k'}$), creating node k' with $u_{k'} = u_k$. However, this means that $\mathcal{B}$ chose $x_{k'}$ itself, and thus i has a Dec-ancestor with a known secret key, meaning that node j was not created in mode (2), which is a contradiction.

Since $\mathcal{B}$ can extract all values d and thus compute $d_{0\to i}$ with $u_i = u^* \cdot g^{d_{0\to i}}$, and since $v_j = v^* \cdot g^{s_j}$, it can translate $\mathrm{DH}(u_i, v_j)$ to $\mathrm{DH}(u^*, v^*)$, as done above.

6 Extensions

In this section, we discuss possible extensions of our construction.

Table 3: Impact on MLS of UPKE/PKE with increasing security (top to bottom). SGM security notions $\mathsf{RealPCFS}_{\mathsf{MLS}}$, $\mathsf{RealPCFS}_{\mathsf{FS+}}$, $\mathsf{FakePCFS}_{\mathsf{MLS}}$ and $\mathsf{FakePCFS}_{\mathsf{FS+}}$ are intuitively described in the text.

Protocol	Security of (U)PKE	Security of MLS using the (U)PKE		
		Restrictions on adv.	Member PCFS	New-member PCFS
MLS_0	PKE, IND-CCA	—	$\mathsf{RealPCFS}_{\mathsf{MLS}}$	$\mathsf{FakePCFS}_{\mathsf{MLS}}$
rTreeKEM	UPKE, IND-CPA [ACDT20]	delivers messages in order, does not inject	$\mathsf{RealPCFS}_{\mathsf{FS+}}$	—
MLS_2	UPKE, CU-CCA [DKW21]	delivers messages in order	$\mathsf{RealPCFS}_{\mathsf{FS+}}$	—
MLS_3^-	UPKE, IND-CCA$^-$ [this work]	—	$\mathsf{RealPCFS}_{\mathsf{FS+}}$	$\mathsf{FakePCFS}_{\mathsf{MLS}}$
MLS_3	UPKE, IND-CCA [this work]	—	$\mathsf{RealPCFS}_{\mathsf{FS+}}$	$\mathsf{FakePCFS}_{\mathsf{FS+}}$

Batch proof verification. We observe that many π proofs for different statements can be verified in a batch, at the cost of only one pairing evaluation. In particular, proofs $\pi_1, \ldots, \pi_\ell \in \hat{\mathbb{G}}$ for statements $u_1, \ldots, u_\ell \in \mathbb{G}$ can be verified by sampling random $r_1, \ldots, r_\ell \in \mathbb{Z}_p$ and checking if $\mathrm{e}(u_1^{r_1} \cdot \ldots \cdot u_\ell^{r_\ell}, h) \overset{?}{=} \mathrm{e}(g, \pi_1^{r_1} \cdot \ldots \cdot \pi_\ell^{r_\ell})$.

This can be particularly useful for MLS, where joiners have to verify a π proof for each public key in the group state (the number of public keys is roughly twice the number of members). However, even current group members have to verify π proofs for multiple public keys, in certain bad scenarios even as many as group members.

Anchor tags. One can somewhat relax how joiner security is defined for UKEM and still end up with a UKEM notion sufficient to prove new-member security in dynamic group protocols like MLS. Instead of forcing $\mathsf{Verify}_{\mathsf{jt}}$ to use an initial public key pk_0 to validate a key pk we could generalize the UKEM syntax to generate arbitrary (but constant size) *anchor tags at* as part of KeyGen. Anchor tags are then used in place of pk_0 in $\mathsf{Verify}_{\mathsf{jt}}$. Of course, we could include such a tag inside pk_0 under the hood, but that can result in sub-optimal efficiency as it means growing (potentially quite significantly) the size of such public keys. By separating the anchor tag from public keys in the syntax, applications can include the anchor tag only when it is really needed.

As it turns out, in our scheme no additional anchor tag beyond pk_0 itself is needed, which is why we decided against using a more complicated syntax. However, explicit anchor tags wouldn't be a problem for the applications we had in mind and the added flexibility may prove helpful for future UKEM constructions.

7 Impact of UPKE on the Security MLS

We discuss the impact of using various UPKE schemes (and their security notions) on MLS. See Table 3 for a summary. Many claims in this section are intuitive; we leave formal definitions and proofs of the security of MLS with UPKE as an open problem.

7.1 SGM Basics

We first recall some facts about Secure Group Messaging (SGM) protocols needed to understand the impact of UPKE. SGM protocols allow members of a dynamic group, i.e., one whose membership can change over time, to securely broadcast messages to the group. For clarity we call these "(application) messages" which stand in contrast to SGM protocol messages which we refer to as "packets". The execution of most SGM protocols proceeds in *epochs* which are characterized by a fixed state; notably, a fixed group membership. Typically (e.g. in MLS) epochs' states also include an implicit *group key* used to encrypt the application messages broadcast by members of the epoch. To modify a group's state, any group member can *create* a new "child" epoch with the new state (e.g. a new set of members) by broadcasting a single packet to the group. Other group members *transition* from the "parent" epoch to the "child" epoch when they receive the packet. SGM protocols like MLS also guarantee *agreement*; namely members can only read each others messages if they agree on the current epoch's state and its preceding history.

If two members, say A and B, each create and transition to a child epoch for the same parent epoch, the result is a *forked* group — the views of A and B now diverge to different branches of the fork and they will no longer be able to process each other's messages as they no longer agree on the group state.

In terms of security, at a minimum, SGM protocols should guarantee confidentiality of the application messages in the presence of an adversary who controls the network and can repeatedly corrupt parties by leaking their secret states.[13] If the adversary corrupts group members, some messages are not confidential. The exact flavor of *Post-Compromise Forward Secrecy* (PCFS) of a protocol determines which messages are guaranteed to be confidential given a sequence of corruptions. For a given execution, protocols with stronger PCFS generally guarantee confidentiality of more messages.

7.2 Comparison of MLS with Different UPKE / PKE Schemes

We distinguish three aspects in which the security of MLS is impacted by the security of the (U)PKE it uses. The first concerns the adversarial capabilities for which MLS can be proven secure. The second concerns the strength of the PCFS afforded parties participating in "real" groups, i.e. groups created by an honest party. The third concerns the PCFS strength for parties in "fake" groups, i.e. groups created arbitrarily by the adversary. (Interestingly, under the right conditions, some MLS-like protocols can indeed ensure security for some epochs in fake groups.)

Baseline. As the baseline for our comparison, we use MLS_0 – the original MLS protocol, which uses PKE. We denote by $\mathsf{RealPCFS}_{\mathsf{MLS}}$ and $\mathsf{FakePCFS}_{\mathsf{MLS}}$, the flavor of PCFS for members of real and fake groups, respectively, achieved by MLS_0. MLS_0 can be proven secure without any restrictions on the adversary, assuming IND-CCA security of the PKE [AJM22].[14]

UPKE schemes from prior work. The rTreeKEM protocol of [ACDT20] introduced the idea of replacing PKE with UPKE in MLS. That work used an IND-CPA style security for UPKE with the added restriction that the adversary cannot produce forks when evolving UPKE keys. The authors prove that rTreeKEM satisfies an SGM security notion which similarly disallows forks. That is, the network is modeled as ideal authenticated network[15] and the adversary must deliver all packets in the same order to all parties. An authenticated network also prevents the adversary from ever creating fake groups. However,

[13] SGM also guarantees other properties like authenticity, but for simplicity we focus on confidentiality.

[14] The analysis of MLS_0 in [AJM22] disallows certain corruptions to avoid the so-called commitment problem, but this is outside the scope of this work.

[15] More accurately, the notion of [ACDT20] disallows only those packet modifications/injections that cannot be prevented by the protocol (using signatures and MACs). As noted in [ACJM20], a more realistic insecure network model allows the adversary to inject on behalf of corrupted parties, which then requires IND-CCA (U)PKE.

in executions with no forks, their flavor of PCFS for real groups, denoted by $\mathsf{RealPCFS}_{\mathsf{FS}+}$, is strictly stronger than $\mathsf{RealPCFS}_{\mathsf{MLS}}$ achieved by MLS_0. The following example MLS execution separates $\mathsf{RealPCFS}_{\mathsf{FS}+}$ from $\mathsf{RealPCFS}_{\mathsf{MLS}}$.

1. A, B and C are in the ("real") group in epoch E_1. C is corrupted.
2. B creates epoch E_2, removing C from the group. A transitions to E_2.
3. B creates epoch E_3 (with the same set of members but a new group key). A transitions to E_3.
4. Adversary corrupts A.

Unlike $\mathsf{RealPCFS}_{\mathsf{MLS}}$, $\mathsf{RealPCFS}_{\mathsf{FS}+}$ implies confidentiality for messages broadcast in E_2. To see why, note that corrupting A lets the adversary learn the secret (U)PKE key of A in epoch E_3. For MLS_0, PKE is used. So, A uses the same leaked secret key while in epoch E_1 to compute the group key for epoch E_2 by processing the packet from sent by B. Thus the adversary can do the same.[16] Meanwhile, in rTreeKEM, which uses UPKE, processing B packets results in A updating her UPKE secret key. So, although A's updated secret key leaked in E_3 this does not reveal the secret key A used to process B's packet announcing epoch E_2.

Following the ideas of [ACDT20] and with an eye towards stronger security of MLS, the work [DKW21] proposed a stronger UPKE security notion (and matching construction) called CU-CCA. Let MLS_2 denote rTreeKEM using a CU-CCA secure UPKE. Intuitively, MLS_2 achieves the same $\mathsf{RealPCFS}_{\mathsf{FS}+}$ as rTreeKEM but over insecure channels instead of authenticated ones.

Nevertheless, CU-CCA is still not sufficient to remove the restriction preventing the adversary from reordering messages, as we show in Appendix A. Moreover, MLS_2 does not provide any meaningful PCFS for fake groups. Indeed, the lack of joiner tags in the UPKE of [DKW21] lets the adversary create fake groups with UPKE keys originally created by honest parties but which then weren't updated correctly. Without joiner tags, parties joining the fake group have no means of verifying the provenance of the updated UPKE keys.

This work. Let MLS_3 denote MLS that uses our IND-CCA secure UPKE in place of PKE and where joining members use joiner tags to validate the provenance of updated UPKE keys. First of all, MLS_3 achieves (at least) the $\mathsf{FakePCFS}_{\mathsf{MLS}}$ flavor of PCS of MLS_0 as well as the same improved $\mathsf{RealPCFS}_{\mathsf{FS}+}$ as rTreeKEM and MLS_2. Moreover, as for MLS_0 (and in contrast to rTreeKEM and MLS_2) this holds for insecure channels. This means that MLS_3 achieves "best of both worlds" security (proving this is outside the scope of this work).

To gain further insight, we observe that all of the above is already true for MLS_3 assuming a weaker UPKE security, call it IND-CCA$^-$, which differs from our IND-CCA in that *no* Rev queries are allowed if JChal is called. Let MLS_3^- denote MLS_3 where the UPKE scheme satisfies IND-CCA$^-$. MLS_3^- already achieves the "best of both worlds" security. On the other hand, MLS_3 has strictly better PCFS than MLS_3^- in fake groups, say $\mathsf{FakePCFS}_{\mathsf{FS}+}$. Indeed, the following example of an MLS execution separates $\mathsf{FakePCFS}_{\mathsf{FS}+}$ from $\mathsf{FakePCFS}_{\mathsf{MLS}}$.

1. A and B are in the "real" group in epoch E_{r1}.
2. Adversary invites C to a "fake" group in epoch E_{f1} with members A, C and the adversary.

[16] More precisely, computing E_2's group key also requires a second value not obtained from B's packet, but the adversary obtains that by corrupting C at the beginning.

3. C creates epoch E_{f2} in the fake group, removing the adversary.
4. B creates epoch E_{r2} in the real group. A transitions to E_{r2} and is corrupted.

In contrast to $\mathsf{FakePCFS}_{\mathsf{MLS}}$, $\mathsf{FakePCFS}_{\mathsf{FS+}}$ implies confidentiality of secret messages broadcasted in E_{f2}.

We start with some intuition about why E_{f2} is secure. Let $pk_{r1}, pk_{r2}, pk_{f1}$ denote A's UPKE keys in E_{r1}, E_{r2}, E_{f1}, respectively. Intuitively, we know that E_{f2} is secure because B honestly updates pk_{r1} to pk_{r2} *after* the adversary creates E_{f1} with pk_{f1}. This means that pk_{f1} used by C is independent of the leaked sk_{r2}. For example, if the adversary chose $pk_{f1} = pk_{r1}$, it is obvious that UPKE security guarantees security of C's message encrypted to pk_{r1} even if sk_{r2} leaks.

Note that the temporal aspect is crucial in this scenario. If B created E_{r2} before the adversary created the fake group (i.e., if Step 4 happened before 2 and 3), E_{f2} would not be secure. Indeed, the adversary could choose pk_{f1} to be the corrupted pk_{r2} (or pk_{f1} is pk_{r2} updated by the adversary in its head, possibly making pk_{f1} and pk_{r2} look unrelated).

We note that security models of [AJM22] and all CGKA/SGM works we are aware of are too coarse to capture this distinction as they do not model temporal relations of events in an execution. Instead, they require considering E_{f2} insecure as there exists a sequence of the transpired events for which E_{f2} would be insecure despite the actual execution not having followed this order.[17]

We next give some intuition as to why IND-CCA$^-$ is sufficient for $\mathsf{FakePCFS}_{\mathsf{MLS}}$. Observe that the regular PKE of MLS_0 can be viewed as UPKE where updates do nothing and the secret/public keys never change. The above clearly does not achieve UPKE member security (when JChal is disabled). However, it does achieve UPKE joiner security of IND-CCA$^-$ (when MChal is disabled), since the static secret key never leaks. Moreover, MLS_0 achieves $\mathsf{FakePCFS}_{\mathsf{MLS}}$ using only PKE. It is therefore perhaps not surprising that MLS_3^- also achieves $\mathsf{FakePCFS}_{\mathsf{MLS}}$ using UPKE with joiner security matching that of PKE.

7.3 UPKE vs Optimal PCFS

While MLS_3 has much better PCFS than MLS_0 (thanks to its use of UPKE), it does not achieve the optimal PCFS defined in [ACJM20]. At a high level, this is because our UPKE is not "optimally secure" in that the adversary (may) be able to undo updates for which it knows the coins used (e.g. ones it generates). In our UPKE security notion, this is reflected by restricting the adversary to not corrupt keys in the dec-closure of the challenge path. HIBE, on the other hand, can easily be used to construct an "optimally secure" UPKE which is why the HIBE-based messaging scheme of [ACJM20] provides optimal PCFS.

The following example of an MLS execution separates MLS_3 from the optimally secure HIBE-based construction of [ACJM20].

1. A, B and C are in the ("real") group in epoch E_1. C is corrupted.
2. The adversary, impersonating C, creates epoch E_2. B transitions to E_2.
3. The adversary corrupts B.
4. A creates epoch E_3, removing the adversary.

Confidentiality of messages broadcast in E_3 is guaranteed by the HIBE-based construction but not by MLS_3. Indeed, let sk_1 and sk_2 denote B's keys in E_1 and E_2, resp. In MLS_3, the adversary corrupts sk_2 and can compute sk_1 by undoing its update from sk_1 to sk_2 that it generated (it subtracts d_2 in our scheme). Now sk_2 allows the adversary to decrypt

[17] We leave a more fine-grained "temporally aware" CGKA/SGM security model for capturing such subtleties for future work.

ciphertexts encrypted by A to pk_1 while creating E_3. Thus the adversary learns the group's secrets in E_3 and can decrypt messages in E_3.

On the other hand, with the HIBE-based construction, B generated a HIBE master secret key sk (in some parent epoch of E_1), sk_1 is below sk in the hierarchy, derived for the identity vector $(\ldots, i_1)$ and sk_2 is underneath sk_1 for the identity vector $(\ldots, i_1, i_2)$ (the messaging protocol makes sure that i_1, i_2, $\ldots$ are unique for the epochs). Now by the security of HIBE, sk_2 reveals no information about sk_1 *even if the adversary chooses* i_2.

Acknowledgments. This work was funded by the Vienna Science and Technology Fund (WWTF) [10.47379/VRG18002] and by the Austrian Science Fund (FWF) [10.55776/F8515-N]. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions.

References

[AAN+22] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. CoCoA: Concurrent continuous group key agreement. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 815–844. Springer, Heidelberg, May / June 2022.

[ABR98] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. DHIES: An encryption scheme based on the Diffie-Hellman problem. Contributions to IEEE P1363a, September 1998.

[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, Heidelberg, April 2001.

[ACDT20] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277. Springer, Heidelberg, August 2020.

[ACDT21] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of MLS. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1463–1483. ACM Press, November 2021.

[ACJM20] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 261–290. Springer, Heidelberg, November 2020.

[AHKM22] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 69–82. ACM Press, 2022.

[AJM22] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 34–68. Springer, Heidelberg, August 2022.

[ALP22] Calvin Abou Haidar, Benoît Libert, and Alain Passelègue. Updatable public key encryption from DCR: efficient constructions with stronger security. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 11–22. ACM, 2022.

[AMT23] Joël Alwen, Marta Mularczyk, and Yiannis Tselekounis. Fork-resilient continuous group key agreement. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 396–429. Springer, 2023.

[APS23] Calvin Abou Haidar, Alain Passelègue, and Damien Stehlé. Efficient updatable public-key encryption from lattices. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part V*, volume 14442 of *LNCS*, pages 342–373. Springer, Singapore, December 2023.

[AW23] Kyoichi Asano and Yohei Watanabe. Updatable public key encryption with strong CCA security: Security analysis and efficient generic construction. *IACR Cryptol. ePrint Arch.*, page 976, 2023.

[BBLW22] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption. RFC 9180, February 2022.

[BBR+23] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 514–532. Springer, Heidelberg, December 2001.

[BLS04] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. On the selection of pairing-friendly groups. In Mitsuru Matsui and Robert J. Zuccherato, editors, *SAC 2003*, volume 3006 of *LNCS*, pages 17–25. Springer, Heidelberg, August 2004.

[Bow] Sean Bowe. Bls12-381: New zk-snark elliptic curve construction.

[CCD$^+$20] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *J. Cryptol.*, 33(4):1914–1983, 2020.

[Dam92] Ivan Damgård. Towards practical public key systems secure against chosen ciphertext attacks. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 445–456. Springer, Heidelberg, August 1992.

[DKW21] Yevgeniy Dodis, Harish Karthikeyan, and Daniel Wichs. Updatable public key encryption in the standard model. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part III*, volume 13044 of *LNCS*, pages 254–285. Springer, Heidelberg, November 2021.

[DP92] Alfredo De Santis and Giuseppe Persiano. Zero-knowledge proofs of knowledge without interaction (extended abstract). In *33rd FOCS*, pages 427–436. IEEE Computer Society Press, October 1992.

[DV19] F. Betül Durak and Serge Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In Nuttapong Attrapadung and Takeshi Yagi, editors, *IWSEC 19*, volume 11689 of *LNCS*, pages 343–362. Springer, Heidelberg, August 2019.

[EJKM22] Edward Eaton, David Jao, Chelsea Komlo, and Youcef Mokrani. Towards post-quantum key-updatable public-key encryption via supersingular isogenies. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021*, volume 13203 of *LNCS*, pages 461–482. Springer, Heidelberg, September / October 2022.

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018.

[FO22] Georg Fuchsbauer and Michele Orrù. Non-interactive mimblewimble transactions, revisited. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part I*, volume 13791 of *LNCS*, pages 713–744. Springer, Heidelberg, December 2022.

[FPS20] Georg Fuchsbauer, Antoine Plouviez, and Yannick Seurin. Blind schnorr signatures and signed ElGamal encryption in the algebraic group model. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 63–95. Springer, Heidelberg, May 2020.

[GHJL17] Felix Günther, Britta Hale, Tibor Jager, and Sebastian Lauer. 0-RTT key exchange with full forward secrecy. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 519–548. Springer, Heidelberg, April / May 2017.

[GM15] Matthew D. Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. In *2015 IEEE Symposium on Security and Privacy*, pages 305–320. IEEE Computer Society Press, May 2015.

[HKP$^+$21] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. A concrete treatment of efficient continuous group key agreement via multi-recipient PKEs. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1441–1462. ACM Press, November 2021.

[JMM19] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 159–188. Springer, Heidelberg, May 2019.

[JS18] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018.

[OP01] Tatsuaki Okamoto and David Pointcheval. The gap-problems: A new class of problems for the security of cryptographic schemes. In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 104–118. Springer, Heidelberg, February 2001.

[PR18] Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 3–32. Springer, Heidelberg, August 2018.

[PS00] David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13(3):361–396, June 2000.

[Sah99] Amit Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In *40th FOCS*, pages 543–553. IEEE Computer Society Press, October 1999.

[SKSW22] Yumi Sakemi, Tetsutaro Kobayashi, Tsunekazu Saito, and Riad S. Wahby. Pairing-Friendly Curves. Internet-Draft draft-irtf-cfrg-pairing-friendly-curves-11, Internet Engineering Task Force, November 2022. Work in Progress.

A Insufficiency of Prior UPKE Notions

Here we present a couple of counterexamples demonstrating the insufficiency of the UPKE security notion introduced in [DKW21] and used in [ALP22, AW23, APS23] for building insider secure dynamic group protocols.

Forking, even without injections. The first counterexample shows that UPKE security of [DKW21] is not sufficient to obtain a dynamic group protocol secure against adversaries who can deliver packets out of order, but *not* to inject ciphertexts.

Let U' be a UPKE scheme satisfying the security notion of [DKW21] with (at most) λ-bit long secret keys. We modify U' as to obtain scheme U as follows. Along with the secret key sk' generated by U', during key generation of U we also sample a uniform random λ-bit string s to be stored along with sk' as part of U's secret key sk. To update the $sk_i = (sk'_i, s_i)$ in U, we update sk'_i to obtain an sk'_{i+1} just as U' does. In addition we flip a fair coin b. If $b = 0$, we set the updated string to $s_{i+1} := H(s)$ for random oracle H. Otherwise, when $b = 1$ we set the updated string to be $s_{i+1} := H(s) \oplus sk'_i$ where $\oplus$ denotes bit-wise XOR.

It is easy U still satisfies [DKW21] UPKE security. After all, the values of s have no effect on the output of any algorithm. Moreover, in the [DKW21] security game the adversary is only allowed to leak one secret key overall. Clearly, a single s does not leak anything.

However, consider an adversary attacking U which can leak the secret keys of t keys pairs, each one the result of a unique update of a fixed target key pair (pk, sk) where $sk = (sk', s)$. With overwhelming probability in t this will reveal both $H(s)$ and $H(s) \oplus sk'$ which means we can get no privacy for anything encrypted to pk. Crucially, this holds despite the adversary only corrupting keys derived from (pk, sk) via honest and secret updates!

Now suppose we use U as our UPKE construction with which we replace PKE in, say, MLS. Most modern security notions in the literature for a dynamic group protocol like MLS [AJM22, AAN+22] allow the adversary to deliver packets in an arbitrary order, causing members to have divergent views. In MLS, like in all messaging protocols with comparable efficiency profiles, some of the secret UPKE keys making up a session's cryptographic state are known and used by multiple parties in the group. One such key, for example, is used for decryption by half the group. By selectively forwarding updates (i.e. ciphertexts) to different subsets in such a group the adversary can cause multiple child keys to be derived off a target UPKE key pair (pk, sk). Therefore, using the above attack strategy the adversary can easily put themselves in a position where they can compute sk without ever corrupting a party that had in its state at the time of corruption (a) sk itself, (b) any of its predecessor of sk nor (c) any other secret key to which sk or one of its predecessors was directly or indirectly encrypted. Under such conditions, we'd expect UPKE's forward secrecy to ensure pk still provides (at least) CPA security. Of course, in this case it doesn't since the adversary can compute sk. Thus, the adversary can use knowledge of sk to derive group keys and decrypt messages sent to the group for periods in the session for which we would otherwise expect security.

Injecting, even without forks. The first counterexample shows that UPKE security of [DKW21] is not sufficient to obtain a dynamic group protocol secure against adversaries who can inject ciphertexts but they *cannot* deliver packets out of order.

Again, let U' be a UPKE scheme satisfying the security notion of [DKW21] with (at most) λ-bit long secret keys. We modify U' as to obtain scheme U as follows. The secret key sk in U consists of sk' generated by U' as well as a uniform random λ-bit string s. To update sk_i, the scheme U updates sk'_i and copies s as is. Finally, a ciphertext in U is of the form $(0, c')$ or $(1, s')$, the latter never outputted by encryption. To decrypt $(0, c')$, the scheme U decrypts c' with sk', and to decrypt $(1, s')$, the scheme U outputs sk' if $s' = s$ or $\perp$ otherwise.

It is easy to see that U satisfies [DKW21] UPKE security, since the latter disallows any decryption queries after any secret key is revealed.[18] Therefore, s is random and independent of the adversary's view as long as decryption queries are allowed, so the chance of guessing it is negligible.

However, suppose U was used in MLS. Say a group member A generates a key pair (s, sk'_0) and pk_0 and sends (s, sk'_0) to another member B (A and B will be able to decrypt). Then another member C sends a message containing a ciphertext c encrypted to pk_0; B receives c but A does not (yet). We expect that the adversary cannot decrypt c even if B is now corrupted, since B already updated the secret key to $sk_1 = (s, sk'_1)$. However, this is not the case — using s from B's key, the adversary can inject $(1, s)$ to A

B APS Security for UPKE

We recall the (long) syntax and CCA-style security notions for UKEM of [APS23] which is based on the analogues UPKE syntax and security introduced in [DKW21] and also used in [ALP22].

Key Generation. $\mathsf{UKeyGen}(1^\kappa) \to (pk, sk)$ outputs a public-secret key pair.

Encapsulation. $\mathsf{UEncaps}(pk) \to (K, c)$ encapsulates key K to public key pk in ciphertext c.

Decapsulation. $\mathsf{UDecaps}(sk, c) \to K/\perp$ decapsulates ciphertext c with secret key sk to obtain key K.

Public Key Update. $\mathsf{UpdatePk}(pk) \to (up, pk')$ takes input an public key pk and outputs a updated public key pk' and update token up.

Secret Key Update. $\mathsf{UpdateSk}(sk, up) \to sk'$ takes input secret key sk and update token up and outputs updated secret key sk'.

Update Verification. $\mathsf{VerifyUpdate}(pk, pk', up) \to 0/1$ takes input public key pk, updated public key pk' and update token up returning either true or false.

In particular, updates are not agnostic as UpdatePk takes a target public key to be updated as input. Indeed, the following correctness notion only requires that UpdateSk correctly update the corresponding secret key to the original target public key.

Definition 7 ((k, δ)-Correctness [APS23]). *Let $(pk, sk) \leftarrow \mathsf{UKeyGen}(1^\kappa)$ and $k \in \mathbb{N}$. For $t < k$ let $(up_{t+1}, pk_{t+1}) \leftarrow \mathsf{UpdatePk}(pk_t)$ and $sk_{t+1} \leftarrow \mathsf{UpdateSk}(sk_t, up)$. A UKEM scheme is* (k, δ)-correct *for $\delta > 0$ if $\forall t \leq k$:*

$$\mathbb{P}\left[\mathsf{UDecaps}(sk_t, c_t) \neq K_t \mid (c_t, K_t) \leftarrow \mathsf{UEncaps}(pk_t)\right] < \delta.$$

[18] The likely reason is that after a key sk_i is revealed, the adversary can decrypt ciphertexts to $sk_i, sk_{i+1}, \ldots$ itself. This reasoning makes sense if only one party holds the secret key; after all, it is supposed to have deleted $sk_0, \ldots, sk_{i-1}$. However, this does not take into account the fact that other out-of-sync parties may still have $sk_0, \ldots, sk_{i-1}$.

Game UKEM k-IND-CU-CCA Security [APS23]

```
Exp^{k-IND-CU-CCA}(A)
    $t = 0$
    $\beta \leftarrow_\$ \mathcal{U}(\{0,1\})$
    $(pk_0, sk_0) \leftarrow \mathsf{UKeyGen}(1^\kappa)$.
    $st \leftarrow \mathcal{A}^{\mathsf{Up},\mathsf{Dec},\mathsf{H}}(pk_0)$
    $(c^*, K^*) \leftarrow \mathsf{UEncaps}(pk_t)$;
    if $\beta = 1$ then
        $K^* \leftarrow \mathcal{U}(\mathcal{K})$
    $pk^{chall} = pk^t$
    $st \leftarrow \mathcal{A}^{\mathsf{Up},\mathsf{Dec},\mathsf{H}}(c^*, st)$
    $r^* \leftarrow \mathcal{U}(\mathcal{R})$
    $(up^*, pk^*) \leftarrow \mathsf{UpdatePk}(pk_t; r^*)$
    $sk^* \leftarrow \mathsf{UpdateSk}(sk_t, up^*)$
    $\beta' \leftarrow \mathcal{A}^{\mathsf{H}}(pk^*, sk^*, c^*, st)$
    $\mathcal{A}$ wins if $\beta = \beta'$
```

```
Oracle Up(pk', up)
    if $\mathsf{VerifyUpdate}(pk_t, pk', up) = 0$ then
        return 0
    $pk_{t+1} \leftarrow pk'$
    $sk_{t+1} \leftarrow \mathsf{UpdateSk}(sk_t, up)$
    $t = t + 1$

Oracle Dec(c)
    if $(pk_t = pk^{chall}) \wedge (c = c^*)$ then
        return $\perp$
    return $\mathsf{UDecaps}(sk_t, c)$
```

Fig. 5: The experiment formalizing k-IND-CU-CCA security of UKEM schemes taken from [APS23].

The UKEM k-IND-CU-CCA security game of [APS23] specified in Figure 5 (and the UPKE analogue in [DKW21, ALP22]) capture CCA-style security with key validation for group members (i.e. they adversarially updated keys). However, the game forces all updates to be applied in sequence which means forks are not captured. Joiner security is also not considered. For example, there is no way to aggregate updates. That means, to use VerifyUpdate to validate a UKEM public key provided by an adversary (as part of a downloaded protocol state) a joiner would, at a syntactic level already, need the full history of updates and intermediary public keys leading starting at some initial public key generated with UKeyGen to the current public key being validated. Unfortunately, for joiner security we need validation of adversarial public keys to use a tag with only a constant size.

C Simulation-Multi-Extraction of Schnorr Proofs

Schnorr signatures are zero-knowledge *proofs of knowledge* (PoK) of the secret key, as we define them in Figure 6. A statement for this proof system consists of the public key u and the signed message m. The proof system PoL used by our construction in Figure 2 is

Construction $\mathsf{Sch}[\mathbb{G}, \mathrm{H}]$

```
Prove((u, m); x)
    $r \leftarrow_\$ \mathbb{Z}_p$ ; $v := g^r$
    $c := \mathrm{H}(v, g^x, m)$
    $s := (r + cx) \bmod p$
    return $\pi := (v, s)$

Verify((u, m), π = (v, s))
    $c := \mathrm{H}(v, u, m)$
    return $g^s = v \cdot u^c$
```

Fig. 6: The Schnorr signature scheme defined over a group $\mathbb{G}$ of order p and hash function H, interpreted as proof system for the NP-relation $\{((u,m),x) \in (\mathbb{G} \times \{0,1\}^*) \times \mathbb{Z}_p \mid u = g^x\}$.

then obtained by setting

$$\mathsf{PoL.Prove}((u,u'),x) := \mathsf{Sch.Prove}((u'/u,(u,u')),x) \quad \text{and}$$
$$\mathsf{PoL.Verify}((u,u'),\tau) := \mathsf{Sch.Verify}((u'/u,(u,u')),\tau)$$

Security requires that (1) proofs can be simulated (by programming the random oracle) without knowledge of a witness (zero-knowledge) [PS00]. Further, (2) a proof proves knowledge of a witness, that is, for an adversary that outputs a valid proof for a statement $(u,m) \in \mathbb{G} \times \{0,1\}^*$, there exists an extractor that can extract x such that $u = g^x$ (knowledge soundness). Extraction was originally shown by *rewinding* the adversary, but this incurs a security loss. Modeling adversaries as algebraic algorithms as in the AGM [FKL18], extraction can be done "straight-line", leading to only a negligible extraction error [FPS20]. (Since an algebraic adversary must output "representations" together with any group element, the extractor can compute the witness from these representations with overwhelming probability.)

Fuchsbauer and Orrù [FO22] show that, in the AGM, Schnorr proofs are *strongly simulation-extractable*. This means that after the adversary has obtained simulated proofs π_i for statements (u_i,m_i) of the adversary's choice, from a "fresh" valid statement/proof pair $((u^*,m^*),\pi^*)$ output by the adversary, a witness can be extracted with overwhelming probability. The "freshness" requirement for the *strong* notion is that $((u^*,m^*),\pi^*) \notin \{((u_i,m_i),\pi_i)\}_i$ (i.e., even returning a new proof for a queried statement is legal).

Since in the security proof of our UPKE scheme we need to extract from several proofs, we extend this notion and allow the adversary to return proofs for $(u_1^*,m_1^*),(u_2^*,m_2^*),\ldots \in \mathbb{G} \times \{0,1\}^*$ and declare the adversary successful if from any of these proofs the extractor fails to extract.

A technical property of their definition [FO22] is that, in order to be useful in a security reduction, the notion needs to be w.r.t. *auxiliary input* one additional group element $a \in \mathbb{G}$ of which neither the adversary nor the extractor know the discrete logarithm (cf. Footnote 11).

Theorem 2. *In the ROM and the AGM with one auxiliary group element, Schnorr proofs of knowledge are strongly simulation-extractable. In particular, let n_s be (an upper bound on) the number of simulated proofs, n_h the number of random oracle queries; let $p = |\mathbb{G}|$ and n the number of output adversarial proofs. Then the simulation error is $\epsilon_{n_s}^{sim} := n_s/(p - n_h - n_s)$ and the multi-extraction error is $\epsilon_n^{ext} = n/p$.*

Proof. Let $\mathbb{G}$ be the group defining the statements, g be a generator and $p = |\mathbb{G}|$. Let $a \leftarrow_{\$} \mathbb{G}$ denote the uniformly sampled auxiliary-input group element. We model H as a random oracle which is controlled by the challenger. We describe how the latter simulates proofs and how it extracts witnesses from fresh adversarial proofs.

Simulation. Let $(u_i,m_i) \in \mathbb{G} \times \{0,1\}^*$ be a statement for which the adversary $\mathcal{A}$ queries a simulated proof. The challenger chooses uniform $c_i, s_i \leftarrow_{\$} \mathbb{Z}_p$, sets

$$v_i := g^{s_i} \cdot u_i^{-c_i} \tag{3}$$

and programs the random oracle H so that $\mathrm{H}(v_i,u_i,m_i) = c_i$. By the choice of c_i and s_i, we have that v_i is uniform and independent; therefore the probability that H has already been defined for (v_i,u_i,m_i) is negligible. If this happens, the challenger aborts and $\mathcal{A}$ wins.

In more detail, let n_h be an upper bound on $\mathcal{A}$'s random oracle queries. Then, by the union bound, the probability that when querying n_s simulated proofs, one of the simulations fails, is upper-bounded by $\epsilon_{n_s}^{\mathrm{sim}} := n_s/(p - n_h - n_s)$.

Extraction. Since simulation queries contain a group element, $\mathcal{A}$ must return its representation. $\mathcal{A}$'s first query (u_1, m_1) is therefore accompanied by (α_1, β_1) s.t. $u_1 = g^{\alpha_1} \cdot a^{\beta_1}$. This answer to the query contains a new group element v_1, thus $\mathcal{A}$'s second simulation query (u_2, m_2) is accompanied by γ', ζ', ρ' with

$$\begin{aligned} u_2 &= g^{\gamma'} \cdot a^{\zeta'} \cdot v_1^{\rho'} \overset{(3)}{=} g^{\gamma'} \cdot a^{\zeta'} \cdot g^{s_1 \rho'} \cdot (g^{\alpha_1} \cdot a^{\beta_1})^{-c_1 \rho'} \\ &= g^{\alpha_2} \cdot a^{\beta_2} \end{aligned}$$

with $\alpha_2 := \gamma' + (s_1 - \alpha_1 c_1)\rho'$ and $\beta_2 := \zeta' - \beta_1 c_1 \rho'$ for $1 \leq i \leq n$.

In general, in an inductive fashion, assume that for every query (u_j, m_j) for $j < i$, we have computed a representation α_j, β_j such that $u_j = g^{\alpha_j} \cdot a^{\beta_j}$. The query responses (v_j, s_j) are thus of the following form, for $c_j = \mathrm{H}(v_j, u_j, m_j)$:

$$v_j = g^{s_j} \cdot u_j^{-c_j} = g^{s_j - \alpha_j c_j} \cdot a^{-\beta_j c_j}. \tag{4}$$

Then from any representation

$$u_i = g^{\gamma''} \cdot a^{\zeta''} \cdot v_1^{\rho_1''} \cdots v_{i-1}^{\rho_{i-1}''}$$

we can recursively derive α_i and β_i so that $u_i = g^{\alpha_i} \cdot a^{\beta_i}$.

Consider proofs (v_i^*, s_i^*) for statements (u_i^*, m_i^*), resp., for $1 \leq i \leq n$, output by $\mathcal{A}$ so that

$$(v_i^*, u_i^*, m_i^*, s_i^*) \neq (v_j, u_j, m_j, s_j) \quad \text{for all } i \text{ and } j, \tag{5}$$

that is, none of the statement/proof pairs was a simulation query/response. If $\mathsf{Sch.Verify}$ on input statement (u_i^*, m_i^*) and proof (v_i^*, s_i^*) returns true then

$$v_i^* \cdot (u_i^*)^{c_i^*} = g^{s_i^*} \quad \text{with} \quad c_i^* = \mathrm{H}(v_i^*, u_i^*, m_i^*) \quad \text{for } 1 \leq i \leq n. \tag{6}$$

For any i, consider the point when $\mathrm{H}(v_i^*, u_i^*, m_i^*)$ gets defined. This must be during a random oracle query made by $\mathcal{A}$, since it cannot have made a simulation query for (u_i^*, m_i^*) answered with v_i^*: as there is only one valid value s_i^*, this would mean that $\mathcal{A}$ returned the oracle's response, i.e., (5) does not hold.

Let q_i be the number of simulation queries made before the random oracle query (v_i^*, u_i^*, m_i^*). Since $\mathcal{A}$ is algebraic, it must accompany the query by representations $(\delta_i, \eta_i, \rho_{i,1}, \ldots, \rho_{i,q_i})$ and $(\gamma_i, \zeta_i, \xi_{i,1}, \ldots, \xi_{i,q_i})$ of v_i^* and u_i^*, respectively, that is,

$$\begin{aligned} v_i^* &= g^{\delta_i} \cdot a^{\eta_i} \cdot \prod_{j=1}^{q_i} v_j^{\rho_{i,j}} \overset{(4)}{=} g^{\delta_i + \sum_{j=1}^{q_i}(s_j - \alpha_j c_j)\rho_{i,j}} \cdot a^{\eta_i - \sum_{j=1}^{q_i} \beta_j c_j \rho_{i,j}} \\ u_i^* &= g^{\gamma_i} \cdot a^{\zeta_i} \cdot \prod_{j=1}^{q_i} v_j^{\xi_{i,j}} \overset{(4)}{=} g^{\gamma_i + \sum_{j=1}^{q_i}(s_j - \alpha_j c_j)\xi_{i,j}} \cdot a^{\zeta_i - \sum_{j=1}^{q_i} \beta_j c_j \xi_{i,j}} \end{aligned} \tag{7}$$

Substituting u_i^* and v_i^* in (6) by the above right-hand sides and grouping the coefficients of g and the a yields

$$\begin{aligned} &a^{\left(\eta_i - \sum_{j=1}^{q_i} \beta_j c_j \rho_{i,j}\right) + c_i^* \left(\zeta_i - \sum_{j=1}^{q_i} \beta_j c_j \xi_{i,j}\right)} \\ &\qquad = g^{s^* - \left(\delta_i + \sum_{j=1}^{q_i}(s_j - \alpha_j c_j)\rho_{i,j}\right) - c_i^* \left(\gamma_i + \sum_{j=1}^{q_i}(s_j - \alpha_j c_j)\xi_{i,j}\right)}. \end{aligned} \tag{8}$$

Assume that the representation of u_i^* in (7) is independent of a, that is

$$\zeta_i - \sum_{j=1}^{q_i} \beta_j c_j \xi_{i,j} \equiv_p 0 \; . \tag{9}$$

Then the extractor can output the witness $\log u_i^* = \gamma_i + \sum_{j=1}^{q_i} (s_j - \alpha_j c_j)\xi_{i,j}$.

Otherwise, the coefficient of c_i^* in the exponent of a in (8) is non-zero. The adversary (implicitly) chose the values α_j, β_j for all $1 \le j \le q_i$ when making simulation (or random oracle) queries *before* making the query $\mathrm{H}(v_i^*, u_i^*, m_i^*)$. Likewise, it must have chosen the values η_i, ζ_i and $\rho_{i,j}, \xi_{i,j}$ for all $1 \le j \le q_i$ before making this query. Therefore, c_i^* is chosen uniformly at random *after* all the values in the exponent of a in (8) are defined, and moreover c_i^* is not multiplied by 0. The probability that the exponent of a in (8) is congruent to 0 modulo p is thus $\frac{1}{p}$. If it is different from 0, the reduction can efficiently compute $\log a$ from (8), and from the representation of u_i^* in (7), it can compute the witness $\log u_i^*$.

If the adversary returns n proofs then the probability that for any of them extraction fails is upper-bounded by $\epsilon_n^{\text{ext}} = n/p$. $\square$