Scalable Deniable Group Key Establishment

(full version)

Kashi Neupane[1], Rainer Steinwandt[2⋆], and Adriana Suárez Corona[2⋆⋆]

[1] Atlanta Metropolitan State College, Atlanta, GA 30310
kneupane@atlm.edu
[2] Florida Atlantic University, Boca Raton, FL 33431
{rsteinwa,asuarezc}@fau.edu

Abstract. The popular Katz-Yung compiler from CRYPTO 2003 can be used to transform unauthenticated group key establishment protocols into authenticated ones. In this paper we present a modification of Katz and Yung's construction which maintains the round complexity of their compiler, but for 'typical' unauthenticated group key establishments adds authentication in such a way that deniability is achieved as well. As an application, a deniable authenticated group key establishment with three rounds of communication can be constructed.
Keywords: Group key establishment, Deniability

1 Introduction

To simplify the design process for a group key establishment protocol, it can be convenient to restrict first to a scenario with a passive adversary, where the problem of authenticating protocol participants does not need to be addressed. Once the protocol has been proven secure in such a setting, a generic construction by Katz and Yung from CRYPTO 2003 allows to achieve security against an active adversary at the cost of one additional round [9]. Basically, this compiler appends nonces, along with sender and receiver identifiers, to all protocol messages and signs all messages with a strongly unforgeable signature scheme. This intuitive construction is round-efficient, but problematic when deniability is added as a design goal: unforgeable signatures in a protocol transcript would have to be explained in a way which does not involve the signing party.

In the two-party setting, deniability has been studied by Di Raimondo et al. [7] and Yao and Zhao [11], for instance. The problem of formalizing deniable key establishment in the group setting has been addressed in [4], where a four-round solution in the random oracle model is presented. In [12], Zhang et al. suggest an alternative formalization of deniability along with a three-round protocol in the standard model. Deniable group key establishment in a setting where the computational power of protocol participants differs is addressed by Chen et al. [6] with a proposal in the random oracle model. Compared to [4] and [12], our definition of deniability limits the adaptivity of the adversary in corrupting users,

⋆ RS was supported by the Spanish *Ministerio de Economía y Competitividad* through the project grant MTM-2012-15167.
⋆⋆ ASC was supported by project MTM2010 - 18370 - C04- 01 and FPU grant AP2007-03141, cofinanced by the European Social Fund.

but unlike [12] we give the adversary access to oracles that reveal session keys and send individual messages, and we do not include secret keys of corrupted users in the simulator's input. From a practical point of view this formalization of deniability seems acceptable, and we present the first general compiler to construct authenticated and deniable group key establishment protocols from 'typical' passively secure constructions.

The compiler we suggest builds on the Katz-Yung construction, but replaces the signature scheme with a suitable use of a ring signature, a message authentication code, and a multiparty key encapsulation. Like the original Katz-Yung construction, our compiler is capable of augmenting every passively secure group key establishment to an actively secure one by adding one more round. Moreover, if the unauthenticated protocol does not make use of long-term secrets—which one would typically expect—the protocol output by our compiler is deniable. In particular, applying our compiler to an unauthenticted two-round protocol as the one described in [9], which builds on work of Burmester and Desmedt [5], results in a deniable and authenticated three-round protocol.

2 Preliminaries

As main technical tools, we will make use of a multiparty key encapsulation along with a suitable data encapsulation mechanism to send an identical message to multiple protocol participants in a confidential manner. For implementing the authentication we also make use of a message authentication code and a ring signature. Here and in the subsequent sections, the security parameter will be denoted by k, and notions like polynomial time or negligible refer to k.

2.1 Multi key encapsulation and symmetric encryption

In [10], Smart introduced the notion of a *multi key encapsulation mechanism (mKEM)*, generalizing key encapsulation to a setting with multiple recipients. A group key establishment by Gorantla et al. [8] makes use of this primitive, and the subsequent definitions follow Gorantla et al.

Definition 1 (multi key encapsulation mechanism).
A multi key encapsulation mechanism (mKEM) *is a triple of polynomial time algorithms* $(\texttt{mKeyGen}, \texttt{mEncaps}, \texttt{mDecaps})$ *as follows:*

- $\texttt{mKeyGen}$ *is probabilistic. Given the domain parameters* $\mathbb{D}$*, it generates a pair of public and secret keys* (pk, dk)*.*
- $\texttt{mEncaps}$ *is probabilistic. Given a (polynomial size) set* $\{pk_1, \ldots, pk_n\}$ *of public keys it generates a pair* (K, C) *where* $K \in \{0,1\}^k$ *is a session key and* C *is an encapsulation of this session key under the public keys* $\{pk_1, \ldots, pk_n\}$*.*
- $\texttt{mDecaps}$ *is deterministic. Given a secret key* dk *and an encapsulation* C*, this algorithm outputs the session key* K *or a special error symbol* $\perp$*.*

We require that for all key pairs (pk_i, dk_i) *generated by* $\texttt{mKeyGen}$ *the implication* $(K, C) = \texttt{mEncaps}(\{pk_1, \ldots, pk_n\}) \Longrightarrow \texttt{mDecaps}_{dk_i}(C) = K$ *holds* $(i = 1, \ldots, n)$*.*

To characterize security of an mKEM, we use a similar approach as in the case of a single recipient.

Definition 2 (IND-CCA security). *An mKEM scheme is* IND-CCA secure *if the advantage of any probabilistic polynomial time adversary $\mathcal{A}$ in the game described in Figure 1 is negligible. Here the* advantage *of an adversary $\mathcal{A}$ is defined as the function* $\mathrm{Adv}_{\mathcal{A}}^{\text{IND-CCA}}(k) = |2 \cdot \Pr[b = b'] - 1|$.

- **Setup:** The challenger $\mathcal{C}$ runs the key generation $\texttt{mKeyGen}$ to obtain n key pairs $(pk_1, dk_1), \ldots, (pk_n, dk_n)$ and hands the public keys $pk_1, \ldots, pk_n$ to $\mathcal{A}$.
- **Phase 1:** The adversary can submit queries to a decapsulation oracle with subsets $\mathcal{P}' \subseteq \{pk_1, \ldots, pk_n\}$ and encapsulations C: given a query $\mathsf{mDecaps}(\mathcal{P}', C)$, the challenger computes $\texttt{mDecaps}_{dk_i}(C)$ for each $pk_i \in \mathcal{P}'$. If the same output K is produced for every $pk_i \in \mathcal{P}'$, then K is returned to $\mathcal{A}$. Otherwise, $\perp$ is returned.
- **Challenge:** The adversary chooses a set $\mathcal{P}^* \subseteq \{pk_1, \ldots, pk_n\}$ and gives it to the challenger $\mathcal{C}$. Then $\mathcal{C}$ selects a bit $b \in \{0, 1\}$ uniformly at random and computes $(K_b, C^*) \leftarrow \texttt{mEncaps}(\mathcal{P}^*)$. Finally, $\mathcal{C}$ selects a key K_{1-b} uniformly at random from the session key space and hands $(\{K_0, K_1\}, C^*)$ to $\mathcal{A}$.
- **Phase 2:** The adversary can submit decapsulation queries as in Phase 1, subject to the condition that no $\mathsf{mDecaps}$-query is made that returns K_b.
- **Guess:** The adversary outputs $b' \in \{0, 1\}$ and wins if and only $b = b'$.

Fig. 1. IND-CCA security of a multi key encapsulation mechanism

To be able to actually encrypt messages in our compiler, we combine an mKEM with a suitable *data encapsulation mechanism*, which we realize as a symmetric encryption scheme offering security in the real-or-random sense (cf. Bellare et al. [1]):

Definition 3 (symmetric encryption scheme). *A symmetric encryption scheme is a triple of polynomial time algorithms* $(\texttt{KeyGen}, \texttt{Enc}, \texttt{Dec})$ *as follows:*

- $\texttt{KeyGen}$ *is probabilistic. Given the security parameter* 1^k*, it generates a secret key* K*.*
- $\texttt{Enc}$ *is probabilistic. Given a secret key* K *and a message* $m \in \{0, 1\}^*$*, this algorithm generates a ciphertext* C*.*
- $\texttt{Dec}$ *is deterministic. Given a ciphertext* C *and a secret key* K*, this algorithm outputs either a message* m *or a dedicated error symbol* $\perp$*.*

We require that $m = \texttt{Dec}_K(\texttt{Enc}_K(m))$ *for all keys* K *and for all* $m \in \{0, 1\}^*$*.*

To characterize security, let $\mathcal{RR}(m, b)$ be a *real-or-random oracle*, i. e., on input $m \in \{0, 1\}^*$ a query $\mathcal{RR}(m, 1)$ simply returns m, whereas a query $\mathcal{RR}(m, 0)$ returns a uniformly at random chosen bitstring of the same length as m. For a probabilistic polynomial time algorithm $\mathcal{A}$, now consider the experiment in Figure 2 where $\mathcal{E}_K(\cdot)$ respectively $\mathcal{D}_K(\cdot)$ is an oracle which applies the encryption algorithm $\texttt{Enc}_K(\cdot)$ resp. the decryption algorithm $\texttt{Dec}_K(\cdot)$ to its input.

Building on this experiment, security in the real-or-random sense is defined by measuring $\mathcal{A}$'s success in correctly identifying the secret b used by the real-or-random oracle:

Definition 4 (ROR-CCA security). *Subsequently, we say that a symmetric encryption scheme* $(\texttt{KeyGen}, \texttt{Enc}, \texttt{Dec})$ *is* secure in the sense of real-or-random

- **Setup:** A secret bit $b \leftarrow \{0,1\}$ is chosen uniformly at random and a secret key $K \leftarrow \mathtt{KeyGen}(1^k)$ is created
- **Challenge:** The adversary $\mathcal{A}$ has unrestricted access to the 'composed oracle' $\mathcal{E}_K(\mathcal{RR}(\cdot, b))$. Further, $\mathcal{A}$ has access to $\mathcal{D}_K(\cdot)$ subject to the restriction that no ciphertexts must be queried to $\mathcal{D}_K(\cdot)$ that have been output by $\mathcal{E}_K(\mathcal{RR}(\cdot, b))$.
- **Guess:** The adversary outputs a bit $b' \in \{0,1\}$.

Fig. 2. ROR-CCA security of a symmetric encryption scheme

(ROR-CCA), *if the advantage* $\mathrm{Adv}^{\mathrm{ror-cca}}_{\mathcal{A}} = \mathrm{Adv}^{\mathrm{ror-cca}}_{\mathcal{A}}(k) :=$

$$\Pr\left[1 \leftarrow \mathcal{A}^{\mathcal{E}_K(\mathcal{RR}(\cdot,1)),\mathcal{D}_K(\cdot)}(1^k) : K \leftarrow \mathtt{KeyGen}(1^k)\right] \\ -\Pr\left[1 \leftarrow \mathcal{A}^{\mathcal{E}_K(\mathcal{RR}(\cdot,0)),\mathcal{D}_K(\cdot)}(1^k) : K \leftarrow \mathtt{KeyGen}(1^k)\right]$$

is negligible for all probabilistic polynomial time algorithms $\mathcal{A}$.

2.2 Message authentication codes and ring signatures

To solve the problem of authentication without jeopardizing deniability, our compiler uses a message authentication code as well as a suitable ring signature.

Definition 5 (message authentication code).

A message authentication code (MAC) *is a tuple* $(\mathtt{MKeyGen}, \mathtt{Tag}, \mathtt{Verify})$ *of polynomial time algorithms as follows:*

- $\mathtt{MKeyGen}$ *is probabilistic. Given the domain parameters* $\mathbb{D}$*, it generates a secret key* K*.*
- $\mathtt{Tag}$ *is probabilistic. Given a message* $m \in \{0,1\}^*$ *and a secret key* K *it generates a message tag* $\theta := \mathtt{Tag}_K(m) \in \{0,1\}^*$ *on* m*.*
- $\mathtt{Verify}$ *is deterministic. Given a message* m*, a secret key* K *and a candidate tag* θ*,* $\mathtt{Verify}$ *returns* 1 *if* θ *is a valid tag for the message* m *and* 0 *otherwise.*

The compiler described below assumes that the message authentication code we employ is strongly unforgeable under adaptive chosen message attacks (cf. [2]). Figure 3 describes the corresponding experiment, where $\mathcal{T}_K(\cdot)$ respectively $\mathcal{V}_K(\cdot)$ describes an oracle which applies $\mathtt{Tag}_K(\cdot)$ respectively $\mathtt{Verify}_K(\cdot)$ to its inputs.

Definition 6 (SUF-CMA security).

A message authentication code $(\mathtt{MKeyGen}, \mathtt{Tag}, \mathtt{Verify})$ *is* strongly unforgeable under adaptive chosen message attacks/secure in the sense of SUF-CMA *if for all probabilistic polynomial time adversaries* $\mathcal{A}$ *the advantage* $\mathrm{Adv}^{\text{suf-cma}}_{\mathcal{A}} = \mathrm{Adv}^{\text{suf-cma}}_{\mathcal{A}}(k) := \Pr[\mathsf{Succ}^{\text{suf-cma}}_{\mathcal{A}}]$ *is negligible. Here* $\mathsf{Succ}^{\text{suf-cma}}_{\mathcal{A}}$ *denotes the event that* $\mathcal{A}$ *wins the experiment in Figure 3.*

- **Setup:** A secret key $K \leftarrow \mathtt{MKeyGen}(\mathbb{D})$ is created
- **Challenge:** The adversary $\mathcal{A}$ has unrestricted access to the tagging oracle $\mathcal{T}_K(\cdot)$ and the verification oracle $\mathcal{V}_K(\cdot)$.
- **Guess:** $\mathcal{A}$ outputs a (message, tag)-pair (m, θ) and wins if and only if $\mathtt{MVer}_K(m, \theta) = 1$ and either m has never beeen queried to $\mathcal{T}_K(\cdot)$ or no query of the form $\mathcal{T}_K(m)$ returned the tag θ.

Fig. 3. SUF-CMA security of a message authentication code

Finally, our compiler uses a ring signature which enables a signer to produce signatures which can be verified successfully under several verification keys.

Definition 7 (ring signature scheme). *A* ring signature scheme *is a tuple of polynomial time algorithms* $(\texttt{RKeyGen}, \texttt{RSign}, \texttt{RVerify})$ *as follows:*

- $\texttt{RKeyGen}$ *is probabilistic. Given the security parameter* k*, it generates a pair of keys* (vk, sk)*, where* vk *is a public verification key and* sk *is its corresponding secret signing key.*
- $\texttt{RSign}$*. Given a message* m*, a polynomial size set (a* ring*) of public verification keys* $\mathcal{R} = \{vk_1, \ldots, vk_n\}$ *and a secret key* sk_s *such that* $vk_s \in \mathcal{R}$*, this algorithm produces a signature* σ*.*
- $\texttt{RVerify}$ *is deterministic. Given a message* m*, a signature* σ *and a ring of public keys* $\mathcal{R}$*, this algorithm returns* 1 *if* σ *is a valid signature for the message* m *with respect to the ring* $\mathcal{R}$*, and* 0 *otherwise.*

We require that for any ring $\mathcal{R}$ *comprised of public verification keys produced by* $\texttt{RKeyGen}$ *and for any message* m *the relation* $\texttt{RVerify}(m, \texttt{RSign}_{sk}(m, \mathcal{R}), \mathcal{R}) = 1$ *holds, where* sk *is the secret key for a verification key* $vk \in \mathcal{R}$*.*

For a ring signature, it is usually expected that the adversary cannot know which user in the ring was the actual signer of a message. A strong form of this design goal is known as *anonymity against full key exposure* [3]:

- **Setup:** The challenger $\mathcal{C}$ runs the key generation $\texttt{RKeyGen}$ n times to obtain key pairs $(vk_1, sk_1), \ldots, (vk_n, sk_n)$ and hands the public keys $vk_1, \ldots, vk_n$ to $\mathcal{A}$.
- **Find:** The adversary can (adaptively) query for signatures on a message m under a ring $\mathcal{R}$ from users with a public key $vk_s \in \mathcal{R} \cap \{vk_1, \ldots, vk_n\}$.[a] The challenger responds with $\texttt{RSign}_{sk_s}(m, \mathcal{R})$. At the end of this phase, $\mathcal{A}$ hands a message m^*, a ring $\mathcal{R}^*$ and two indices i_0, i_1 to $\mathcal{C}$, such that $vk_{i_0}, vk_{i_1} \in \mathcal{R}^* \cap \{vk_1, \ldots, vk_n\}$.
- **Challenge:** The challenger $\mathcal{C}$ chooses a bit $b \in \{0, 1\}$ uniformly at random, computes a signature $\texttt{RSign}_{sk_{i_b}}(m^*, \mathcal{R}^*)$, and hands this signature to $\mathcal{A}$ along with the random coins used to generate the key pairs $(vk_1, sk_1), \ldots, (vk_n, sk_n)$.[b]
- **Guess:** The adversary outputs a guess b' for b and wins if and only if $b = b'$.

[a] Note that only vk_s must be chosen from $vk_1, \ldots, vk_n$.
[b] In particular, $\mathcal{A}$ can recover the secret keys $sk_1, \ldots, sk_n$.

Fig. 4. RSIG-ANO: anonymity against full key exposure

Definition 8 (anonymity). *A ring signature scheme is* anonymous against full key exposure *if the advantage of any probabilistic polynomial time adversary* $\mathcal{A}$ *in the game described in Figure 4 is negligible. Here the* advantage *of an adversary* $\mathcal{A}$ *is defined as* $\mathrm{Adv}^{\text{rsig-ano}}_{\mathcal{A}}(k) = \left|\Pr[b = b'] - \frac{1}{2}\right|$.

Of course, as for other kinds of digital signatures, for a ring signature scheme we also expect an appropriate form of existential unforgeability. More specifically, we impose the following.

Definition 9 (RSIG-UF security). *A ring signature scheme is called* unforgeable with respect to insider corruption *if for any probabilistic polynomial time adversary* $\mathcal{A}$ *the advantage* $\mathrm{Adv}^{\text{rsig-uf}}_{\mathcal{A}}(k) := \Pr[\mathsf{Succ}^{\text{rsig-uf}}_{\mathcal{A}}]$ *in the game described in Figure 5 is negligible. Here* $\mathsf{Succ}^{\text{rsig-uf}}_{\mathcal{A}}$ *denotes the event that* $\mathcal{A}$ *wins the experiment in Figure 5.*

– **Setup:** The challenger $\mathcal{C}$ runs the algorithm $\texttt{KeyGen}$ n times to obtain key pairs $(vk_1, sk_1), \ldots, (vk_n, sk_n)$ and hands the public keys $vk_1, \ldots, vk_n$ to $\mathcal{A}$.
– **Challenge:** The adversary is allowed to ask (adaptively) queries for:
  • private keys for a public key vk_i ($i \in \{1, \ldots, n\}$). The challenger returns sk_i.
  • signatures for a message m and a ring $\mathcal{R}$ under a secret key sk_s such that $vk_s \in \mathcal{R} \cap \{vk_1, \ldots, vk_n\}$. The challenger responds with $\texttt{RSign}_{sk_s}(m, \mathcal{R})$.[a]
– **Guess:** The adversary outputs a tuple $(\mathcal{R}^*, m^*, \sigma^*)$ and wins if and only if all of the following hold:
  • $\texttt{RVerify}(m^*, \sigma^*, \mathcal{R}^*) = 1$;
  • in the challenge phase, no secret key sk_i for a $vk_i \in \mathcal{R}^*$ has been queried;
  • in the challenge phase, the pair $(m^*, \mathcal{R}^*)$ was not part of a signature query.

[a] Note that only vk_s must be chosen from $vk_1, \ldots, vk_n$.

Fig. 5. RSIG-UF security of a ring signature

3 Security model

To formalize secure group key establishment, we follow Katz and Yung [9].

3.1 Security goals: semantic security and authentication

Protocol participants. We model the protocol participants as a finite (polynomial size) set of users $\mathcal{U} = \{U_0, \ldots, U_{n-1}\}$. Each user is modeled as a probabilistic polynomial time algorithm, and can execute a polynomial number of protocol instances Π_U^s concurrently ($s \in \mathbb{N}$). To describe a protocol instance Π_U^s, seven variables are associated with it:

acc_U^s**:** a boolean variable, which is set to TRUE if and only if the session key stored in sk_U^s has been accepted;
pid_U^s**:** stores the identities of those users in $\mathcal{U}$ with which a key is to be established (including U);
sid_U^s**:** stores a non-secret session identifier that can be used as public reference to the session key stored in sk_U^s;
sk_U^s**:** stores the session key and is initialized with a distinguished NULL value;
state_U^s**:** stores state information;
term_U^s**:** a boolean variable, which is set to TRUE if and only if the protocol execution has terminated;
used_U^s**:** indicates if this instance is involved in a protocol run.

Initialization. Before the actual protocol execution, an initialization phase without adversarial interference takes place. In this phase, for each user $U \in \mathcal{U}$ a long-term secret key can be established and public keys can be distributed among users.

Remark 1. When considering passive adversaries only, typically no long-term secrets are required and this initialization phase becomes trivial.

Communication network and adversarial capabilities. The network is non-private and fully asynchronous. Arbitrary point-to-point connections among the users are possible (no dedicated broadcast functionality is assumed to be available). The adversary $\mathcal{A}$ is modeled as probabilistic polynomial time algorithm with complete control over the communication network. Its capabilities are captured by these *oracles*:

- $\mathsf{Send}(U_i, s_i, M)$: sends the message M to instance $\Pi_{U_i}^{s_i}$ of user U_i and returns the protocol message output by that instance after receiving M. The Send oracle also enables $\mathcal{A}$ to initialize a protocol execution: sending the special message $M = \{U_{i_1}, \ldots, U_{i_r}\}$ to an unused instance $\prod_{U_i}^{s_i}$ initializes a protocol run among $U_{i_1}, \ldots, U_{i_r} \in \mathcal{U}$. After such a query, $\prod_{U_i}^{s_i}$ sets $\mathsf{pid}_{U_i}^{s_i} := \{U_{i_1}, \ldots, U_{i_r}\}$, $\mathsf{used}_{U_i}^{s_i} :=$ TRUE, and processes the first step of the protocol.
- $\mathsf{Execute}(U_1, s_1, \ldots, U_r, s_r)$: returns a complete protocol transcript among the specified unused instances.
- $\mathsf{Reveal}(U, s)$: returns the session key sk_U^s if $\mathsf{acc}_U^s =$ TRUE and a NULL value otherwise.
- $\mathsf{Corrupt}(U)$: for a user $U \in \mathcal{U}$ this query returns U's long term secret-key.

An adversary with access to all of the above oracles is considered to be *active*. A *passive* adversary is not granted access to the Send oracle. In addition to the mentioned oracles, $\mathcal{A}$ has access to a Test oracle, which can be queried only once: the query $\mathsf{Test}(U, s)$ can be made with an instance Π_U^s that has accepted a session key. Then a bit $b \leftarrow \{0, 1\}$ is chosen uniformly at random; for $b = 0$, the session key stored in sk_U^s is returned, and for $b = 1$ a uniformly at random chosen element from the space of session keys is returned. We consider only *correct* group key establishments, and our correctness definition follows [9].

Definition 10 (correctness). *A group key establishment is* correct *if for all instances $\Pi_i^{s_i}$, $\Pi_j^{s_j}$ which have accepted with $\mathsf{sid}_i^{s_i} = \mathsf{sid}_j^{s_j}$ and $\mathsf{pid}_i^{s_i} = \mathsf{pid}_j^{s_j}$, the condition $\mathsf{sk}_i^{s_i} = \mathsf{sk}_j^{s_j} \neq$ NULL holds.*

To define what we mean by a secure group key establishment, we rely on the following notion of *partnering*:

Definition 11 (partnering). *Two terminated instances $\prod_{U_i}^{s_i}$ and $\prod_{U_j}^{s_j}$ with $U_i \neq U_j$ are* partnered *if both $\mathsf{sid}_{U_i}^{s_i} = \mathsf{sid}_{U_j}^{s_j}$ and $\mathsf{pid}_{U_i}^{s_i} = \mathsf{pid}_{U_j}^{s_j}$.*

Remark 2. According to Definition 11, different instances of the same user cannot be partnered. This implies that no deterministic protocol can be secure (cf. [9] for a discussion).

Having fixed what we mean by partnered instances, we can now specify which instances can be queried to the Test oracle:

Definition 12 (freshness with forward secrecy). *An instance $\prod_{U_i}^{s_i}$ is said to be* fresh *if none of the following events has occured:*

– *the adversary queried* $\mathsf{Corrupt}(U_j)$ *for some* $U_j \in \mathsf{pid}_i^{s_i}$ *before a query of the form* $\mathsf{Send}(U_l, s_l, *)$ *has taken place where* $U_l \in \mathsf{pid}_i^{s_i}$*;*
– *the adversary queried* $\mathsf{Reveal}(U_i, s_i)$*;*
– *the adversary queried* $\mathsf{Reveal}(U_j, s_j)$ *for an instance* $\prod_{U_j}^{s_j}$ *that is partnered with* $\prod_{U_i}^{s_i}$*.*

If forward secrecy is not a concern, the following weaker definition of freshness can be considered, and the subsequent results hold with either definition of freshness:

Definition 13 (freshness without forward secrecy). *An instance* $\prod_{U_i}^{s_i}$ *is said to be* fresh *if none of the following events has occurred:*

– *the adversary queried* $\mathsf{Corrupt}(U_j)$ *for some* $U_j \in \mathsf{pid}_i^{s_i}$ *before a query of the form* $\mathsf{Send}(U_l, s_l, *)$ *has taken place where* $U_l \in \mathsf{pid}_i^{s_i}$*;*
– *the adversary queried* Corrupt *after a query to* Execute, Reveal, Send, *or* Test*;*
– *the adversary queried* $\mathsf{Reveal}(U_i, s_i)$*;*
– *the adversary queried* $\mathsf{Reveal}(U_j, s_j)$ *for an instance* $\prod_{U_j}^{s_j}$ *that is partnered with* $\prod_{U_i}^{s_i}$*.*

We write $\mathsf{Succ}_{\mathcal{A}}$ for the event that the adversary $\mathcal{A}$ queries Test with a fresh instance and correctly guesses the random bit b used by the Test oracle and refer to $\mathrm{Adv}_{\mathcal{A}}^{\mathrm{ke}} = \mathrm{Adv}_{\mathcal{A}}^{\mathrm{ke}}(k) := \left|\Pr[\mathsf{Succ}] - \frac{1}{2}\right|$ as *advantage* of $\mathcal{A}$. The starting point for the compiler below is an unauthenticated, secure group key establishment protocol, i. e., a protocol where the adversary is assumed to be passive:

Definition 14 (semantic security). *A group key establishment protocol is* (semantically) secure*, if* $\mathrm{Adv}_{\mathcal{A}}^{\mathrm{ke}} = \mathrm{Adv}_{\mathcal{A}}^{\mathrm{ke}}(k)$ *is negligible for all passive probabilistic polynomial time adversaries* $\mathcal{A}$*.*

A first goal of the compiler is to add authentication, i. e., to establish security guarantees against an active adversary:

Definition 15 (semantic security & authentication). *A key establishment protocol is* authenticated and (semantically) secure*, if* $\mathrm{Adv}_{\mathcal{A}}^{\mathrm{ke}} = \mathrm{Adv}_{\mathcal{A}}^{\mathrm{ke}}(k)$ *is negligible for all active probabilistic polynomial time adversaries* $\mathcal{A}$*.*

3.2 A privacy goal: deniability

In addition to authentication and semantic security, the compiler discussed in the next section aims at the resulting protocol to be deniable.

Deniability according to [4] Let $\mathcal{A}_d$ denote a probabilistic polynomial time algorithm with the security parameter 1^k as input. In addition, $\mathcal{A}_d$ obtains the public information pk made available in the initialization phase as input (after application of the compiler below this includes in particular the public verification keys of the underlying ring signature scheme). Finally $\mathcal{A}_d$ obtains as input an upper bound q_c on the number of protocol participants that can be corrupted.

After having obtained this input, $\mathcal{A}_d$ interacts with protocol instances via the Corrupt, Reveal, and Send oracle as usual[3]. However, $\mathcal{A}_d$ must not query Test, and at most q_c queries to Corrupt can be submitted. Eventually, $\mathcal{A}_d$ outputs a bitstring $T_{\mathcal{A}_d} = T_{\mathcal{A}_d}(k, q_c, pk)$—which represents a protocol transcript that serves as evidence for the involvement of a particular user in the group key establishment. We denote by $T_{\mathcal{A}_d} = T_{\mathcal{A}}(k, q_c)$ the random variable that describes $T_{\mathcal{A}_d}(k, q_c, pk)$ with the randomness for $\mathcal{A}_d$, for protocol instances, and in the initialization phase, being chosen uniformly at random.

To capture deniability a second algorithm $\mathcal{S}_d$, to which we refer as *simulator*, is used: this simulator accepts the same input as $\mathcal{A}_d$ and can impose the same maximum number q_c of corrupted users as the latter. However, $\mathcal{S}_d$ is not allowed to invoke any uncorrupted user. More specifically, $\mathcal{S}_d$ can submit up to q_c queries to Corrupt, but can neither query Reveal nor Send (nor Execute nor Test). The output of $\mathcal{S}_d$ is a bitstring $T_{\mathcal{S}_d}(k, q_c, pk)$, and analogously as for $\mathcal{A}_d$ we define a random variable $T_{\mathcal{S}_d}(k, q_c)$, based on uniformly at random chosen randomness.

Definition 16 (deniability). *A group key establishment protocol is* deniable *if for every probabilistic polynomial time adversary $\mathcal{A}_d$ as specified above and every $q_c \in \mathbb{N}_0$ there is a probabilistic polynomial time simulator $\mathcal{S}_d$ such that $T_{\mathcal{A}_d}(k, q_c)$ and $T_{\mathcal{S}_d}(k, q_c)$ are computationally indistinguishable. In other words, no probabilistic polynomial time algorithm can distinguish $T_{\mathcal{A}_d}(k, q_c)$ and $T_{\mathcal{S}_d}(k, q_c)$ with non-negligible probability.*

A more relaxed notion of deniability The definition of deniability just discussed allows the adversary $\mathcal{A}_d$ to fix the corrupted parties in a fully adaptive manner. In the definition used subsequently we restrict this freedom and require $\mathcal{A}_d$ to complete all corruptions before querying Send. On the intuitive side, this materializes the idea that the parties who are willing to conspire (and reveal their secret keys to this aim) already enter protocol executions with this intent. As we still allow an arbitrary subset of the users to be corrupted, the resulting notion of deniability seems still natural and useful.

Remark 3. Unlike [12], we do not integrate authentication into the definition of deniability. Further, differing from [12], we give the adversary used in the definition of deniability full access to Send and Reveal and do not include secret keys of corrupted users in the simulator's input.

As before, let $\mathcal{A}_d$ denote a probabilistic polynomial time algorithm with the security parameter 1^k and public information pk from the initialization phase as input. No upper bound on the number of corruptions is imposed. In a first phase $\mathcal{A}_d$ has access to the Corrupt-oracle only, and can (adaptively) corrupt an arbitrary subset of the users (including the case of no user or all users being corrupted). Hereafter, in a second phase, $\mathcal{A}_d$ interacts with the protocol participants via the Reveal- and Send-oracle. Neither Corrupt nor Test may be

[3] In particular, queries to Send can be used to simulate the Execute oracle, so Execute can be omitted.

queried in this phase. Analogously as in the definition of [4], $\mathcal{A}_d$ outputs a bitstring $T_{\mathcal{A}_d} = T_{\mathcal{A}_d}(k, pk)$ to evidence the involvement of a particular user in the group key establishement. Let $T_{\mathcal{A}_d} = T_{\mathcal{A}_d}(k)$ be the random variable describing $T_{\mathcal{A}_d}(k, pk)$ with the randomness for $\mathcal{A}_d$, for all protocol instances, and in the initialization phase being chosen uniformly at random.

The simulator $\mathcal{S}_d$ obtains the same input as $\mathcal{A}_d$, but can only access the Corrupt oracle—no access to Reveal, Send, or Test is available. The output of $\mathcal{S}_d$ is a bitstring $T_{\mathcal{S}_d}(k, pk)$, and analogously as for $\mathcal{A}_d$ we define a random variable $T_{\mathcal{S}_d}(k)$ based on uniformly at random chosen randomness. Consider the following experiment for a probabilistic polynomial time distinguisher $\mathcal{X}$ outputting 0 or 1: the challenger flips a random coin $b \in \{0, 1\}$ uniformly at random. If $b = 1$, the transcript $T_{\mathcal{A}_d}(k)$ is handed to $\mathcal{X}$, whereas for $b = 0$ the transcript $T_{\mathcal{S}_d}(k)$ is handed to $\mathcal{X}$. The distinguisher $\mathcal{X}$ wins whenever the guess b' it outputs for b is correct; the advantage of $\mathcal{X}$ is denoted by $\mathrm{Adv}^{\mathrm{den}}_{\mathcal{X}} := \left|\Pr[b = b'] - \frac{1}{2}\right|$.

In this paper we will use the following definition of deniability:

Definition 17 ((relaxed) deniability). *A group key establishment protocol is* deniable *if for every polynomial time adversary $\mathcal{A}_d$ as specified above there exists a probabilistic polynomial time simulator $\mathcal{S}_d$ such that the following holds:*

- *With overwhelming probability, the number of* Corrupt*-queries of $\mathcal{S}_d$ is less than or equal to the number of* Corrupt*-queries of $\mathcal{A}_d$.*
- *For each probabilistic polynomial time distinguisher $\mathcal{X}$, the advantage* $\mathrm{Adv}^{\mathsf{den}}_{\mathcal{X}}$ *in the above experiment is negligible.*

4 From unauthenticated to authenticated and deniable

Subsequently we denote by (mKeyGen, mEncaps, mDecaps) an IND-CCA secure multi key encapsulation (see Definition 2) and by (KeyGen, Enc, Dec) a ROR-CCA secure symmetric encryption scheme (see Definition 4). To simplify the description, we assume that $\mathtt{KeyGen}(1^k)$ simply returns a uniformly at random chosen bitsting in $\{0, 1\}^k$ (alternatively one could use keys obtained from mDecaps to fix the randomness of KeyGen). By (MKeyGen, Tag, Verify) we denote an SUF-CMA secure message authentication code (see Definition 6) and by (RKeyGen, RSign, RVerify) an RSIG-UF secure ring signature scheme (see Definition 9) which is anonymous in the sense of Definition 8.

4.1 Description of the proposed compiler

The proposed compiler modifies a given (semantically secure) *unauthenticated* group key establishment protocol P to obtain a protocol P' which is *authenticated*. Moreover, if the original protocol P does not make use of long-term secrets, then the resulting protocol P' is *deniable*. Further, if the original protocol is forward secure the compiled protocol preserves this property. For the ease of notation, assume that $U_0, \ldots, U_{n-1}$ are the users who want to establish a secret key. One of the protocol participants has a special role in the compiled protocol—it is the only user that will sign a message in the newly added Round 0. We refer to this user as initiator and without loss of generality assume that U_0

plays this role. Moreover, for the ease of notation, in our description, we do not explicitly refer to individual instances.

Finally, for the messages in protocol P, let $m_{i,l}^{(j)}$ be the message sent by user U_i in the j-th round to user U_l. We can without loss of genereality assume that instead of sending these messages, in Round j the user U_i broadcasts the combined message $m_{i,j} = U_i||j||\left(m_{i,1}^{(j)}, \ldots, m_{i,n-1}^{(j)}\right)$. In particular, $m_{i,j}$ includes an (unprotected) identifier of the sender U_i of the message and of the round number. Each recipient U_l can recover $m_{i,l}^{(j)}$ in the obvious way, and this change does neither affect the security nor the round complexity of P.

Initialization phase. In addition to the initialization for protocol P, each user U_i generates a (public key, secret key)-pair (pk_i, dk_i) for the above-mentioned multi key encapsulation scheme, and the public keys are made available to all users (and the adversary). Similarly, each user U_i generates a (verification key, signing key)-pair (vk_i, sk_i) for the before-mentioned ring signature scheme.

Next, our compiler adds a new round to protocol P as follows:

Introduction of Round 0. In this new initial round, each user U_i ($i \neq 0$):

- chooses a random nonce $r_i \in \{0,1\}^k$.
- broadcasts $U_i||0||r_i$.

The initiator U_0 performs the following steps:

- run $\texttt{MKeyGen}$ to generate a key K_0 for the message authentication code;
- produces a ring signature $\sigma := \texttt{RSig}_{sk_0}(K_0||\mathsf{pid}_0||U_0||0||r_0, \mathsf{pid}_0)$;
- computes $(K, C) \leftarrow \texttt{mEncaps}(\mathsf{pid}_0)$;
- produces a ciphertext $E := \texttt{Enc}_K(K_0||\mathsf{pid}_0||U_0||0||r_0||\sigma)$ using the symmetric key K;
- computes a tag $\mathsf{tag}_0 = \texttt{Tag}_{K_0}(C, E)$, and
- broadcasts $U_0||0||(C, E)||\mathsf{tag}_0$.

After receiving the Round 0 message of all parties, each user U_i executes the following steps:

- set $\mathsf{nonces}_{U_i} = ((U_1, r_1), \cdots, (U_n, r_n))$ and store this value;
- run $\texttt{mDecaps}_{dk_i}(C)$ to obtain K;
- decrypt the ciphertext E using K;
- verify the ring signature for the ring pid_i and the tag tag_0; if the verification fails or if $\mathsf{pid}_0 \neq \mathsf{pid}_i$, the protocol is aborted.

Now, in each original round of P we use K_0 for authentication as follows:

Modification of Round j, $j \neq 0$. If the protocol is not aborted, if user U_i is supposed to broadcast $m_{i,j}$ in protocol P, then U_i will instead do the following:

- use K_0 to compute a tag $\mathsf{tag}_{i,j} = \texttt{Tag}_{K_0}(m_{i,j}||\mathsf{nonces}_{U_i})$.
- broadcasts $m_{i,j}||\mathsf{tag}_{i,j}$.

When receiving a message $m_{l,j}||\mathsf{tag}_{l,j}$, user U_i checks the following:

- $U_l \in \mathsf{pid}_0$
- j is the expected round number

– Verify the tag $\mathsf{tag}_{l,j}$.

If any of these checks fails, the protocol is aborted without accepting a session key. Otherwise, the session identifier is the concatenation of all messages sent and received by the protocol instance during its execution and the session key is as in P.

Remark 4. With a slight abuse of notation, here we identify a partner identifier pid_j with the set of public keys of the users contained in this partner identifier.

4.2 Security analysis

Making no further assumptions about the protocol P, we have the following result, which says that the above compiler adds authentication as desired:

Proposition 1. *With the above notation, the group key establishment obtained from the compiler in Section 4.1 is authenticated and secure in the sense of Definition 15 (in particular, forward security is preserved).*

Proof. We prove the security of the protocol by 'game hopping', letting the probabilistic polynomial time adversary $\mathcal{A}$ of the compiled protocol P' interact with a simulator $\mathcal{S}$. The success probability respectively the advantage of $\mathcal{A}$ in Game i will be denoted by $\Pr[\mathsf{Succ}_{\mathcal{A}}^{\text{Game } i}]$ respectively $\mathrm{Adv}_{\mathcal{A}}^{\text{Game } i}$. By q_{send} we denote a polynomial upper bound for the number of queries by $\mathcal{A}$ to the Send-query, and analogously we write q_{execute} for a polynomial upper bound on the number of queries to $\mathsf{Execute}$ made by $\mathcal{A}$.

Game 0: This game is identical to the original attack game, with all oracles of the adversary being simulated faithfully by $\mathcal{S}$. Hence, $\mathrm{Adv}_{\mathcal{A}}^{\mathsf{ke}} = \mathrm{Adv}_{\mathcal{A}}^{\text{Game } 0}$.

Game 1: Let Repeat be the event that some user U_i uses a nonce r_i in Round 0 which this user has used previously already. This game is identical to Game 0 with the only exception that we abort the simulation and consider $\mathcal{A}$ as successful whenever the event Repeat occurs. We have $\Pr[\mathsf{Repeat}] \leq (q_{\text{send}} + n \cdot q_{\text{execute}})^2/2^k$, and hence $|\mathrm{Adv}_{\mathcal{A}}^{\text{Game } 0} - \mathrm{Adv}_{\mathcal{A}}^{\text{Game } 1}| \leq \frac{(q_{\text{send}}+n\cdot q_{\text{execute}})^2}{2^k}$ is negligible.

Game 2: In this game the simulator makes a uniform at random guess which instance will be queried to Test by $\mathcal{A}$ and also guesses which instance will initiate the corresponding protocol execution. Whenever such a guess turns out to be incorrect, we abort and consider the adversary $\mathcal{A}$ as successful. Otherwise this game is identical to the previous one. As $\mathcal{A}$ can involve at most $q_{\text{send}} + n \cdot q_{\text{execute}}$ instances, we have $\mathrm{Adv}_{\mathcal{A}}^{\text{Game } 2} \leq (q_{\text{send}} + n \cdot q_{\text{execute}})^2 \cdot \mathrm{Adv}_{\mathcal{A}}^{\text{Game } 1}$, and it will suffice to recognize $\mathrm{Adv}_{\mathcal{A}}^{\text{Game } 2}$ as negligible.

Game 3: Let $\mathsf{ForgeRS}$ be the event that, for the Test-instance, $\mathcal{A}$ succeeds in forging a new ring signature for the initiator U_0 in Round 0 before querying $\mathsf{Corrupt}(U_i)$ for some $U_i \in \mathsf{pid}_0$. Game 3 is identical to Game 2 with the only exception that we abort the simulation and consider $\mathcal{A}$ as successful whenever the event $\mathsf{ForgeRS}$ occurs. Occurence of this event yields immediately an adversary $\mathcal{A}_{\text{rsig}}$ against the ring signature scheme, and $|\mathrm{Adv}_{\mathcal{A}}^{\text{Game } 3} - \mathrm{Adv}_{\mathcal{A}}^{\text{Game } 2}| \leq \mathrm{Adv}_{\mathcal{A}_{\text{rsig}}}^{\text{rsig-uf}}$.

Game 4: This game is identical to Game 3 except that $\mathcal{S}$ produces the ciphertext E under an encryption of a freshly generated key $K' \leftarrow \mathtt{KeyGen}(1^k)$ instead of using the real key K. To bound $|\mathrm{Adv}_{\mathcal{A}}^{\text{Game 4}} - \mathrm{Adv}_{\mathcal{A}}^{\text{Game 3}}|$ we derive from $\mathcal{S}$ an algorithm $\mathcal{A}_{\text{mkem}}$ to attack the IND-CCA security of the underlying mKEM: $\mathcal{A}_{\text{mkem}}$ runs a simulation of $\mathcal{S}$ as in Game 3 and uses as $\mathcal{P}^*$ the public keys of the users in the Test-instance's partner identifier. The only modification in the simulation of $\mathcal{S}$ is for the initator U_0: denoting by $(\widetilde{K}_0, \widetilde{K}_1, C')$ the triple obtained from the mKEM challenger, $\mathcal{A}_{\text{mkem}}$ replaces (K, C) with $(\widetilde{K}_0, C')$ in Round 0 and uses $\widetilde{K}_0$ to compute the ciphertext E. Whenever $\mathcal{A}$ correctly identifies the session key after receiving the challenge of the (simulated) Test-oracle, $\mathcal{A}_{\text{mkem}}$ outputs the guess $b' = 0$, whenever $\mathcal{A}$ guesses incorrectly, $\mathcal{A}_{\text{mkem}}$ outputs $b' = 1$.
Writing b^{ind} and b^{Test} for the values of the random bit used by the mKEM challenger and the random bit of the (simulated) Test-oracle, respectively, we get (with a slight abuse of notation) $\Pr[\mathsf{Succ}_{\mathcal{A}_{\text{mkem}}}^{\text{IND-CCA}}] =$

$$
\begin{aligned}
&1/2 \cdot \Big(\Pr[1 \leftarrow \mathcal{A}^{b^{\mathsf{Test}}=1} \mid b^{\mathsf{ind}} = 0]/2 + (1 - \Pr[1 \leftarrow \mathcal{A}^{b^{\mathsf{Test}}=0} \mid b^{\mathsf{ind}} = 0])/2\Big) \\
&+ 1/2 \cdot \Big((1 - \Pr[1 \leftarrow \mathcal{A}^{b^{\mathsf{Test}}=1} \mid b^{\mathsf{ind}} = 1])/2 + \Pr[1 \leftarrow \mathcal{A}^{b^{\mathsf{Test}}=0} \mid b^{\mathsf{ind}} = 1]/2\Big) \\
&= 1/4 \cdot \Big(\Pr[\mathsf{Succ}_{\mathcal{A}}^{\text{Game 3}}] - \Pr[\mathsf{Succ}_{\mathcal{A}}^{\text{Game 4}}]\Big) + 1/2
\end{aligned}
$$

Consequently, $2 \cdot \mathrm{Adv}_{\mathcal{A}_{\text{mkem}}}^{\text{IND-CCA}} = 2 \cdot |2 \cdot \Pr[\mathsf{Succ}_{\mathcal{A}_{\text{mkem}}}^{\text{IND-CCA}}] - 1| =$

$$\big|\Pr[\mathsf{Succ}_{\mathcal{A}}^{\text{Game 3}}] - \Pr[\mathsf{Succ}_{\mathcal{A}}^{\text{Game 4}}]\big| \geq \big|\mathrm{Adv}_{\mathcal{A}}^{\text{Game 4}} - \mathrm{Adv}_{\mathcal{A}}^{\text{Game 3}}\big|.$$

Game 5: This game is identical to Game 4 except that in Round 0 of the protocol, $\mathcal{S}$ replaces the ciphertext E, sent by the initiator U_0 of the Test-instance, with an encryption of a uniformly at random chosen bitstring of the appropriate length. To bound $|\mathrm{Adv}_{\mathcal{A}}^{\text{Game 5}} - \mathrm{Adv}_{\mathcal{A}}^{\text{Game 4}}|$ we can derive from $\mathcal{A}$ an algorithm $\mathcal{A}_{\text{ror}}$ to attack the ROR-CCA security of the underlying symmetric encryption scheme: $\mathcal{A}_{\text{ror}}$ runs a simulation of Game 4 with the following exception: to create the ciphertext E for the initiator U_0 of the Test-instance, the 'composed oracle' $\mathcal{E}_K(\mathcal{RR}(\cdot, b))$ is invoked, and for decrypting messages other than E under K, the decryption oracle $\mathcal{D}_K(\cdot)$ is invoked. The $\mathsf{Corrupt}$, $\mathsf{Execute}$, Reveal, Send and Test oracle for $\mathcal{A}$ can be simulated by $\mathcal{A}_{\text{ror}}$ in the obvious way.
Whenever $\mathcal{A}$ correctly identifies the session key after receiving the challenge of the (simulated) Test-oracle, $\mathcal{A}_{\text{ror}}$ outputs 1, i. e., claims that its encryption oracle operates in 'real mode', whenever $\mathcal{A}$ guesses incorrectly, $\mathcal{A}_{\text{ror}}$ outputs 0. We obtain $|\mathrm{Adv}_{\mathcal{A}_{\text{ror}}}^{\text{ror-cca}}| = \big|\Pr[\mathsf{Succ}_{\mathcal{A}}^{\text{Game 4}}] - \Pr[\mathsf{Succ}_{\mathcal{A}}^{\text{Game 5}}]\big| \geq \big|\mathrm{Adv}_{\mathcal{A}}^{\text{Game 5}} - \mathrm{Adv}_{\mathcal{A}}^{\text{Game 4}}\big|$.

Game 6: Let $\mathsf{ForgeMAC}$ denote the event that $\mathcal{A}$ succeeds in forging a new valid (message, tag)-pair for a user U_i before quering $\mathsf{Corrupt}(U_j)$ for user some $U_j \in \mathsf{pid}_j$. This game is identical to Game 5 with the only exception that we abort the simulation and consider $\mathcal{A}$ as successful whenever

the event ForgeMAC occurs. Occurence of this event yields immediately an adversary $\mathcal{A}_{\text{mac}}$ against the message authentication code, and $|\text{Adv}_{\mathcal{A}}^{\text{Game 6}} - \text{Adv}_{\mathcal{A}}^{\text{Game 5}}| \leq \text{Adv}_{\mathcal{A}_{\text{mac}}}^{\text{suf-cma}}$.

To conclude the proof, we show $\text{Adv}_{\mathcal{A}}^{\text{Game 6}}$ is negligible by showing that the simulator $\mathcal{S}$ can, running a simulation of $\mathcal{A}$, be turned into a (passive) adversary against P. To attack P, first of all $\mathcal{S}$ corrupts each user in protocol P to obtain all long-term secrets. Moreover, S generates all (public key, secret key)-pairs for the ring signature scheme and for the mKEM and therewith has the long-term secrets of all parties in protocol P'. Making use of this information and maintaining a 'nonces and transcripts list' NT, the simulator $\mathcal{S}$ can run a simulation of Game 6 for $\mathcal{A}$ with oracle queries being answered as follows:

- **Execute queries.** When $\mathcal{A}$ makes a query Execute which does not involve the Test-instance, then $\mathcal{S}$ executes the protocol itself, and stores the resulting nonces nonces_{U_i} along with a special symbol $\perp$ to indicate an empty transcript in the list NT. Knowing all long-term secrets, this simulation of Execute is perfect.
  If the Test-instance is involved, $\mathcal{A}$ queries its own Execute-oracle, and completes the resulting transcript T of protocol P in the obvious way to obtain a perfect simulation of Game 6, making use of the known long-term secrets of all parties. The respective nonces along with the 'padded' transcript T' are stored in the list NT.
- **Send queries.** We denote a query which initiates a new execution for an instance of a user U_i by Send_0. The second Send query to the same instance which includes the message of the form (U_j, r_j) for each user in the pid_i is denoted by Send_1. On a query $\mathsf{Send}_0(U_i, *)$, the simulator $\mathcal{S}$ chooses a random nonce $r_i \in \{0,1\}^k$ itself and replies with the respective Round 0 message. On a Send_1 query that is not directed to the Test-instance, $\mathcal{S}$ computes nonces_{U_i}, stores $(\mathsf{nonces}_{U_i}, \perp)$ in the List NT, and answers by computing the next step of P'.
  If $\mathcal{S}$ receives a Send_1-query for the Test-instance, then $\mathcal{S}$ first looks in its list NT for an entry of the form $(\mathsf{nonces}_{U_i}, T')$. If such an entry exists, then $\mathcal{S}$ takes the appropriate message from T' and answers to $\mathcal{A}$. If such an entry does not exist, then $\mathcal{S}$ queries its Execute-oracle, produces the transcript of P', stores T' in the list NT as it was done previously and answers $\mathcal{A}$ with the appropriate message.
- **Reveal queries.** From Game 2, $\mathcal{S}$ knows the Test-instance, and hence can answer all queries to Reveal by computing the session key by itself.
- **Corrupt queries.** Since $\mathcal{S}$ knows all long-term secrets of users in the protocol P', $\mathcal{S}$ replies in the obvious way.
- **Test query.** From Game 2, $\mathcal{S}$ knows the Test-instance, and can simply forward $\mathcal{A}$'s query to its own Test-oracle.

In summary, the simulator $\mathcal{S}$ answers all queries of $\mathcal{A}$ exactly as in Game 6, and whenever $\mathcal{A}$ violates the semantic security of P', then $\mathcal{S}$ violates the semantic security of P: $\text{Adv}_{\mathcal{A}}^{\text{Game 6}} \leq \text{Adv}_{\mathcal{S}}^{\text{ke}}$. □

In principle we can apply the compiler in Section 4.1 to some fully authenticated protocol, which signs all messages sent by parties. In such a case we cannot expect that the compiled protocol is deniable. The more typical passively secure protocol does not involve any long-term secrets, and in such a setting the proposed compiler does ensure deniability:

Proposition 2. *If the group key establishment* P *does not involve long-term secrets, then the group key establishment* P' *obtained by applying the compiler in Section 4.1 to* P *is deniable in the sense of Definition 17.*

Proof. We prove the deniability of the protocol by 'game hopping', letting the probabilistic polynomial time adversary $\mathcal{A}_d$ of the compiled protocol P' and the simulator $\mathcal{S}_d$ interact with the challenger $\mathcal{C}$. The advantage of the distinguisher $\mathcal{X}$ in Game i will be denoted by $\mathrm{Adv}_{\mathcal{X}}^{\text{Game } i}$.

Game 0: This game is identical to the original deniability game, with all oracles of the adversary and simulator being simulated faithfully by $\mathcal{C}$. Consequently, $\mathrm{Adv}_{\mathcal{X}}^{\mathrm{den}} = \mathrm{Adv}_{\mathcal{X}}^{\text{Game } 0}$.

Game 1: This game is identical to Game 0 except that in the simulation of the Send-oracle, $\mathcal{C}$ changes Round 0 messages for the initiator U_0 if no participant $U_i \in \mathsf{pid}_0$ has been corrupted (i. e., the adversary does not have any private key (sk_i, dk_i)). In this case, to compute the ciphertext E, the simulator encrypts just a random bitstring of the appropriate length.
To argue that the advantage of the distinguisher $\mathcal{X}$ in Game 0 and Game 1 differs only negligibly, consider the following adversary $\mathcal{D}$ against the real-or-random indistinguishability of the symmetric encryption scheme: the algorithm $\mathcal{D}$ will take the role of $\mathcal{C}$ and act as challenger for $\mathcal{A}_d$ and $\mathcal{S}_d$. In addition, $\mathcal{D}$ runs $\mathcal{X}$ as a subroutine. To initiate $\mathcal{A}_d$ and $\mathcal{S}_d$, the adversary $\mathcal{D}$ uses parameters he creates (honestly) on his own. Further, $\mathcal{D}$ simulates all of $\mathcal{A}_d$'s and $\mathcal{S}_d$'s protocol instances faithfully, except when answering queries of the form $\mathsf{Send}(U_0, s_0, \mathsf{pid}_0)$ from $\mathcal{A}_d$ to initiate the protocol with a partner identifier that involves no corrupted users. To answer the latter, he uses the real-or-random oracle instead of the encryption algorithm for determining $E = \mathtt{Enc}_K(K_0||\mathsf{pid}_0||U_0||0||r_0||\sigma)$. He will use the output of this oracle to answer the query. Finally, $\mathcal{D}$ chooses $b \in \{0, 1\}$ uniformly at random.
If $\mathcal{X}$ outputs a correct guess $b' = b$, then $\mathcal{D}$ outputs 1, otherwise $\mathcal{D}$ outputs 0. Therefore, $\left|\mathrm{Adv}_{\mathcal{X}}^{\text{Game } 0} - \mathrm{Adv}_{\mathcal{X}}^{\text{Game } 1}\right| \leq |\mathrm{Adv}_{\mathcal{D}}^{\mathrm{ror-cca}}(k)|$, which is negligible.

Game 2: This game is identical to Game 1 except that $\mathcal{C}$ changes the simulation of the Send-oracle for Round 0 messages for the initiator U_0 if some participant $U_j \in \mathsf{pid}_0$ has been corrupted. In this case, the adversary has a secret key pair (sk_j, dk_j) (if he has more than one secret key pair, he selects one at random). The challenger faithfully simulates all computations of U_0 using sk_j to compute the required ring signature. If the distinguisher $\mathcal{X}$ notices the difference between this simulation and the one in Game 2, he could be used as blackbox to attack the anonymity of the ring signature:
Let $\mathcal{F}$ be the following adversary against the anonymity of the ring signature: the algorithm $\mathcal{F}$ will act as challenger for $\mathcal{A}_d$ and $\mathcal{S}_d$, and also run a simulation of $\mathcal{X}$. Further, $\mathcal{F}$ is given the public keys $vk_1, \ldots, vk_n$, and initiates $\mathcal{A}_d$

and $\mathcal{S}_d$ with these keys—he creates the other parameters on his own. The algorithm $\mathcal{F}$ chooses an instance Π_U^s and an initiator $U_{i_0} \in \mathsf{pid}_U^s$ at random and simulates all of $\mathcal{A}_d$'s and $\mathcal{S}_d$'s instances faithfully, except when answering the query $\mathsf{Send}(\mathsf{pid}_U^s, U_{i_0})$ to initiate the protocol in the selected instance. In order to answer this query, $\mathcal{F}$ runs $\mathtt{MKeyGen}$ to get a key K_0. He will hand its challenger the message $m^* = K_0||\mathsf{pid}_U^s||U_{i_0}||0||r_0$, the ring $\mathcal{R}^* = \mathsf{pid}_U^s$ and two indices i_0 (the initiator) and i_1 such that $vk_{i_0}, vk_{i_1} \in \mathcal{R} \cap \{vk_1, \ldots, vk_n\}$. Hereafter, $\mathcal{F}$ will receive a challenge σ^*, and $\mathcal{F}$ will simulate all instances faithfully, except when the answer includes $\mathtt{RSign}_{sk_0}(K_0||\mathsf{pid}_U^s||U_{i_0}||0||r_0)$. In this case, he will substitute this value by σ^*. Notice $\mathcal{F}$ can query $\mathsf{Corrupt}$ and RSign for different messages, so he can simulate all other messages and $\mathsf{Corrupt}$ queries faithfully. Finally, $\mathcal{F}$ chooses $b \in \{0,1\}$ uniformly at random. If $\mathcal{X}$ outputs a correct guess $b' = b$, then $\mathcal{F}$ outputs 0, otherwise $\mathcal{F}$ outputs 1. Therefore, $\left|\mathrm{Adv}_{\mathcal{X}}^{\text{Game 1}} - \mathrm{Adv}_{\mathcal{X}}^{\text{Game 2}}\right| \leq 2 \cdot \mathrm{Adv}_{\mathcal{F}}^{\text{rsig}-\text{ano}}(k)$, which is negligible. Notice that the simulation provided now to $\mathcal{A}_d$ by the challenger $\mathcal{C}$ is the same $\mathcal{S}_d$ would provide, thus the distinguisher's advantage in this case is 0. Collecting all advantages, we get: $\mathrm{Adv}_{\mathcal{X}}^{\text{den}} \leq |\mathrm{Adv}_{\mathcal{D}}^{\text{ror-cca}}(k)| + 2 \cdot \mathrm{Adv}_{\mathcal{F}}^{\text{rsig-ano}}(k)$, which is negligible. □

5 Conclusion

Given an unauthenticated group key establishment, the protocol compiler we described outputs an authenticated group key establishment. In terms of round complexity one additional round is needed, just as in the Katz-Yung compiler. As a privacy feature, however, our compiler ensures deniability, provided that the given unauthenticated protocol does not involve long-term secrets (which can normally be expected). Applying our compiler to an unauthenticated two-round protocol such as the one described in [9], which builds on work of Burmester and Desmedt [5], yields a deniable group key establishment with three rounds. In summary, our compiler seems a quite interesting alternative to the popular Katz-Yung construction, if privacy guarantees are a concern.

References

1. M. Bellare, A. Desai E. Jokipii, and P. Rogaway. A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation, August 1997. Full paper of an extended abstract that appeared in the *Proceedings of the 38th Symposium on Foundations of Computer Science, IEEE, 1997.*
2. M. Bellare and C. Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In T. Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *LNCS*, pages 531–545. Springer, 2000.
3. A. Bender, J. Katz, and R. Morselli. Ring Signatures: Stronger Definitions, and Constructions Without Random Oracles. In S. Halevi and T. Rabin, editors, *Theory of Cryptography – TCC 2006*, volume 3876 of *LNCS*, pages 60–79. Springer, 2006.

4. J.-M. Bohli and R. Steinwandt. Deniable Group Key Agreement. In P. Q. Nguyen, editor, *Progress in Cryptology – VIETCRYPT 2006*, volume 4341 of *LNCS*, pages 298–311. Springer, 2006.
5. M. Burmester and Y. Desmedt. A Secure and Efficient Conference Key Distribution System. In A. De Santis, editor, *Advances in Cryptology – EUROCRYPT '94*, volume 950 of *LNCS*, pages 275–286. Springer, 1995.
6. S. Chen, Q. Cheng, and C. Ma. A Deniable Group Key Exchange Protocol for Imbalanced Wireless Networks. In B. Hu, X. Li, and J. Yan, editors, *5th International Conference on Pervasive Computing and Applications (ICPCA) 2010*, pages 1–5. IEEE, 2010.
7. M. Di Raimondo, R. Gennaro, and H. Krawczyk. Deniable Authentication and Key Exchange. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 400–409. ACM, 2006.
8. M. Choudary Gorantla, C. Boyd, J. M. González Nieto, and M. Manulis. Generic One Round Group Key Exchange in the Standard Model. In D. Lee and S. Hong, editors, *Information Security and Cryptology – ICISC 2009*, volume 5984 of *LNCS*, pages 1–15. Springer, 2010.
9. J. Katz and M. Yung. Scalable Protocols for Authenticated Group Key Exchange. In D. Boneh, editor, *Advances in Cryptology – CRYPTO'03*, volume 2729 of *LNCS*, pages 110–125. Springer, 2003.
10. N. P. Smart. Efficient Key Encapsulation to Multiple Parties. In C. Blundo and S. Cimato, editors, *Security in Communication Networks SCN 2004*, volume 3352 of *LNCS*, pages 208–219. Springer, 2005.
11. A. C. Yao and Y. Zhao. Deniable Internet Key Exchange. In J. Zhou and M. Yung, editors, *Applied Cryptography and Network Security – ACNS 2010*, volume 6123 of *LNCS*, pages 329–348. Springer, 2010.
12. Y. Zhang, K. Wang, and B. Li. A Deniable Group Key Establishment Protocol in the Standard Model. In J. Kwak, R. H. Deng, Y. Won, and G. Wang, editors, *Information Security, Practice and Experience – ISPEC 2010*, volume 6047 of *LNCS*, pages 308–323. Springer, 2010.