Scalable Protocols for Authenticated Group Key Exchange

JONATHAN KATZ*   MOTI YUNG†

August 13, 2003

Abstract

We consider the fundamental problem of authenticated group key exchange among n parties within a larger and insecure public network. A number of solutions to this problem have been proposed; however, all provably-secure solutions thus far are not scalable and, in particular, require $O(n)$ rounds. Our main contribution is the first *scalable* protocol for this problem along with a rigorous proof of security in the standard model under the DDH assumption; our protocol uses a constant number of rounds and requires only $O(1)$ "full" modular exponentiations per user. Toward this goal and of independent interest, we first present a scalable compiler that transforms any group key-exchange protocol secure against a passive eavesdropper to an *authenticated* protocol which is secure against an active adversary who controls all communication in the network. This compiler adds only one round and $O(1)$ communication (per user) to the original scheme. We then prove secure — against a passive adversary — a variant of the two-round group key-exchange protocol of Burmester and Desmedt. Applying our compiler to this protocol results in a provably-secure three-round protocol for *authenticated* group key exchange which also achieves forward secrecy.

1 Introduction

Protocols for authenticated key exchange (AKE) allow a group of parties within a larger and completely insecure public network to establish a common secret key (a *session key*) and furthermore to be guaranteed that they are indeed sharing this key with *each other* (i.e., with their intended partners). Protocols for securely achieving AKE are fundamental to much of modern cryptography and network security. For one, they are crucial for allowing symmetric-key cryptography to be used for encryption/authentication of data among parties who have no alternate "out-of-band" mechanism for agreeing upon a common key. Furthermore, they are instrumental for constructing "secure channels" on top of which higher-level protocols can be designed, analyzed, and implemented in a modular manner. Thus, a detailed understanding of AKE — especially the design of provably-secure protocols for achieving it — is critical.

The case of 2-party AKE has been extensively investigated within the cryptographic community (e.g., [23, 11, 24, 9, 31, 6, 35, 19, 20, 21] and others) and is fairly well-understood;

*jkatz@cs.umd.edu. Department of Computer Science, University of Maryland, College Park, MD.
†moti@cs.columbia.edu. Department of Computer Science, Columbia University, New York, NY.

furthermore, a variety of efficient and provably-secure protocols for 2-party AKE are known. Less attention has been given to the important case of *group* AKE where a session key is to be established among $n > 2$ parties; we survey relevant previous work in the sections that follow. Group AKE protocols are essential for applications such as secure video- or tele-conferencing, and also for collaborative (peer-to-peer) applications which are likely to involve a large number of users. The recent foundational papers of Bresson, et al. [16, 14, 15] (building on [9, 10, 7]) were the first to present a formal model of security for group AKE and the first to give rigorous proofs of security for particular protocols. These represent an important initial step, yet much work remains to be done to improve the efficiency and scalability of existing solutions.

1.1 Our Contributions

Rigorous security proofs are essential in this domain; as noted in Section 1.2, many protocols without such proofs were subsequently shown to have subtle flaws allowing potential attacks. It is also clearly desirable for group AKE protocols to be efficient *even for a large number of users*. (In Section 1.3, we discuss in detail the complexity measures we use to evaluate the efficiency of protocols in this setting.) Unfortunately, theoretical work in this area has lagged behind the demand for such protocols in practice; we may summarize the prior "state-of-the-art" for *provably-secure* group AKE protocols as follows (we exclude here centralized protocols in which a designated group manager is assumed; such asymmetric, non-contributory schemes place an unfairly high burden on one participant who is a single point of failure and who must also be trusted to properly generate keys):

- The most efficient solutions are those of Bresson, et al. [16, 14, 15] which adapt previous work of Steiner, et al. [37]. Unfortunately, these protocols do not scale well: to establish a key among n participants, they require n rounds and additionally require (for some players) $O(n)$ "full" modular exponentiations and $O(n)$ communication.

- Subsequent to the present work, a constant-round protocol for group AKE has been proven secure *in the random oracle model*[1] [13]. This protocol does *not* achieve forward secrecy (cf. [24]), and the exposure of a user's long-term secret key exposes all session keys previously-generated by this user. The protocol is also not symmetric, and the initiator must perform $O(n)$ encryptions and send $O(n)$ communication.

Interestingly, the above solutions are the best-known *provably-secure* protocols even under various relaxations of the problem. For example, we are aware of no previous constant-round protocol with a full proof of security in the standard (i.e., non-random oracle) model even for the weaker case of security against a passive eavesdropper (but see footnote 2). Clearly, an $O(n)$-round protocol is not scalable and is unacceptable when the number of parties grows large or when communication between parties is the performance bottleneck.

[1]The random oracle model [8] assumes a public random function to which all parties (including the adversary) have access. This random function is instantiated using a cryptographic hash function (e.g., SHA-1), appropriately modified to have the desired domain and range. Although proofs of security in this model provide heuristic evidence for the security of a given protocol, there exist schemes which are secure in the random oracle model but *insecure* for any concrete instantiation of the random oracle [18].

Our main result is the first *constant-round* and *fully-scalable* protocol for group AKE which is provably-secure in the standard model (i.e., without assuming the existence of "random oracles"). Security is proved via reduction to the decisional Diffie-Hellman (DDH) assumption using the same security model as other recent work in this area [16, 14, 15, 13]. Our protocol also achieves forward secrecy in the sense that exposure of principals' long-term secret keys does not compromise the security of previous session keys. We of course also require that exposure of multiple session keys does not compromise the security of unexposed session keys; see the formal model in Section 2. Our 3-round protocol remains practical even for large groups: it requires each user to send only $O(1)$ communication and to compute only 3 "full" modular exponentiations and $O(n \log n)$ multiplications, generate 2 signatures, and perform $O(n)$ signature verifications.

The difficulty of analyzing protocols for group AKE has led to a number of *ad hoc* approaches to this problem and has seemingly hindered the development of practical and provably-secure solutions (this is evidenced by the many published protocols later found to be flawed; cf. the attacks given in [34, 13]). To manage this complexity, we propose and analyze a scalable *compiler* for this setting which enables a modular approach and therefore greatly simplifies the design and analysis of group AKE protocols. Our compiler transforms any group key-exchange protocol which is secure against a *passive* eavesdropper to one which is secure against a stronger — and more realistic — *active* adversary who controls all communication in the network. If the original protocol achieves forward secrecy, the compiled protocol does too. A compiler with similar functionality has been proposed previously for the 2-party case [6]; however, as discussed in Section 1.2, our compiler scales better as the number of participants grows large (interestingly, our compiler is slightly more efficient that that of [6] even in the 2-party case).

As an additional contribution, we investigate the security of the well-known Burmester-Desmedt protocol [17] for *unauthenticated* group key exchange.[2] Adapting their work, we present a 2-round group key-exchange protocol and rigorously prove its security — against a passive adversary — under the DDH assumption. Applying our above-mentioned compiler to this protocol gives our main result.

1.2 Survey of Previous Work

Group key exchange. A number of works have considered extending the 2-party Diffie-Hellman protocol [23] to the multi-party setting [26, 36, 17, 37, 4, 29, 30]. Most well-known among these are perhaps the works of Ingemarsson, et al. [26], Burmester and Desmedt [17], and Steiner, et al. [37]. These works all assume a passive (eavesdropping) adversary, and only the last of these provides a rigorous proof of security (but see footnote 2).

Authenticated protocols are designed to be secure against the stronger class of adversaries who —in addition to eavesdropping — control all communication in the network (cf. Section 2). A number of protocols for authenticated group key exchange have been suggested [27, 12, 2, 3, 38]; unfortunately, none of these works present rigorous security proofs

[2]Since no proof of security appears in [17], the Burmester-Desmedt protocol has often been considered "heuristic" and not provably-secure (see, e.g., [16, 13]). Subsequent to our work we became aware that a proof of security for a variant of the Burmester-Desmedt protocol (in a weaker model than that considered here) appears in the pre-proceedings of Eurocrypt '94 [22]. See Section 4 for further discussion.

and thus confidence in these protocols is limited. Indeed, attacks on some of these protocols have been presented [34], emphasizing the need for rigorous proofs in a well-defined model. Tzeng and Tzeng [39] prove security of a group AKE protocol using a non-standard adversarial model and assuming a reliable broadcast channel (an assumption we do not make here). Their protocol does not achieve forward secrecy, and an explicit attack on their protocol has recently been identified [13].

Provably-secure protocols. Bresson, et al. [16, 14, 15] have recently given the first formal model of security for group AKE and the first provably-secure protocols for this setting. Their model builds on earlier work of Bellare and Rogaway in the 2-party setting [9, 10] as extended by Bellare, et al. [7] to handle (among other things) forward secrecy. Their provably-secure protocols are all based on the protocols of Steiner, et al. [37], and require $O(n)$ rounds to establish a key among a group of n users. The initial work [16] deals with the static case, and shows a protocol which is secure (and achieves forward secrecy) under the CDH assumption in the random oracle model (as noted there, however, the protocol can also be proven secure in the standard model using the DDH assumption). Unfortunately, the given proof of security applies only for groups of *constant* size.

Later work [14, 15] focuses on the dynamic case where users join or leave and the session key must be updated whenever this occurs. Although we do not explicitly address this issue, note that dynamic group membership can always be handled by running the group AKE protocol from scratch among members of the new group. For the case when individual members join and leave, the complexity of the protocol given here is only slightly worse than the Join and Remove algorithms of [14, 15]. The protocol given here performs even better relative to [14, 15] when merging two sizable groups, or when partitioning a group into two groups of roughly equal size (see the interesting work of Amir, et al. [1] for detailed performance comparisons). Yet, handling dynamic membership even more efficiently remains an interesting topic for future research.

More recently (in work subsequent to ours), a constant-round group AKE protocol with a security proof in the random oracle model has been shown [13]. The given protocol does not provide forward secrecy; in fact (as noted by the authors) an explicit attack is possible when long-term secret keys are exposed. Furthermore, the protocol is not symmetric but instead requires a "group leader" to perform $O(n)$ encryptions and send $O(n)$ communication each time a group key is established.

Compilers for key-exchange protocols. A modular approach such as that used here has previously been used in the design and analysis of key-exchange protocols. Mayer and Yung [33] give a compiler which converts any 2-party protocol into a centralized (non-contributory) group protocol; their compiler invokes the original protocol $O(n)$ times, however, and is therefore not scalable. In work with similar motivation as our own, Bellare, et al. [6] show a compiler which converts unauthenticated protocols into authenticated protocols in the 2-party setting. Their compiler was not intended for the group setting and does not scale as well as ours; extending [6] to the group setting gives a compiler which triples the number of rounds and furthermore requires n signature computations/verifications and an $O(n)$ increase in communication per player per round. In contrast, the compiler presented here adds only a *single* round and introduces an overhead of 1 signature computation, n signature verifications, and $O(1)$ communication per player per round. (In fact, the compiler introduced here is slightly more efficient than that of [6] even in the 2-party case.)

1.3 Complexity Measures for Group AKE Protocols

In this work, we measure the efficiency of group AKE protocols in a number of ways. Although the various complexity measures are adapted from standard work in the context of distributed algorithms [32], there are some differences which we discuss below. This is followed by a comparison of the efficiency of our main protocol with previous work.

Round complexity. The *round complexity* (also known as the *latency* or *time complexity*) of a protocol is simply the number of rounds until the protocol terminates. Note, however, that this measure must be more carefully defined in the presence of an active adversary who controls the scheduling of all messages in the network (indeed, the very notion of a "round" is not well-defined in this model, and an adversary can make the time complexity arbitrarily large by refusing to deliver any messages). Therefore, we measure the round complexity assuming the "best-case" scenario: an adversary who delivers all messages intact to the appropriate recipient(s) as soon as they are sent. This seems to be the only sensible way to measure round complexity in our model.

Message complexity. The *message complexity* of a protocol is typically defined as the total number of messages sent (regardless of their length) by all parties in the course of protocol execution. We deviate from this definition in two ways. First, we define the message complexity in terms of the maximum number of messages sent *by any single user*; this seems to be more useful in determining the scalability of a protocol.[3] Second (and perhaps more controversial), when measuring the message complexity we assume that sending the *same* message to multiple parties incurs the same cost as sending that message to a single party; equivalently, we assume a "broadcast channel" and measure the number of messages a player sends to that channel. This provides a better model in some cases: for example, in wireless networks all messages are broadcast by default and no "point-to-point" channels truly exist. We stress, however, that this abstraction of a broadcast channel is used only for measuring the complexity, and is not assumed to provide the full functionality of a broadcast channel; in particular, an active adversary still has complete control over all communication in the network, and can deliver different messages to different parties. Further discussion appears at the beginning of Section 3.1.

Communication complexity. The *communication complexity* (sometimes called the *bit complexity*) measures the total number of bits communicated throughout the protocol; in contrast to the message complexity, the lengths of the messages are now taken into account. As in the case of message complexity, we are interested here in the maximum communication sent by any single player in the protocol (rather than the total communication), and we make the same "broadcast" assumption as before.

Computational complexity. Finally, the *computational complexity* of a protocol is defined as the maximum amount of computation done by any player in the protocol. Here,

[3]Thus, under our definition a protocol in which each of n players sends a single message has message complexity 1 while a protocol in which a single player sends n messages has message complexity n (even though the total number of messages in each case is the same). Note that the former protocol scales better than the latter if the players are more resource-limited than the network infrastructure.

	Rounds	Messages (per user)		Communication (per user)	
		Point-to-point	Broadcast	Point-to-point	Broadcast
[16]	$O(n)$	2	2	$O(n)$	$O(n)$
[13]	1	$O(n)$	1	$O(n^2)$	$O(n)$
Here	3	$O(n)$	3	$O(n)$	$O(1)$

Table 1: Complexity of provably-secure protocols for group AKE in terms of the number of parties n. Note that [16] proves security only for constant n, and [13] proves security in the random oracle model and does not achieve forward secrecy. Our protocol achieves forward secrecy and is proven secure in the standard model for $n = \mathsf{poly}(k)$.

we are interested in the number of cryptographic operations (e.g., "full" exponentiations or public-key encryptions) performed since these operations are likely to dominate.

With the above in mind, we compare in Table 1 the complexity of our main protocol to that of previous work [16, 13] (recall that our main protocol is obtained by applying the compiler of Section 3 to the group KE protocol of Section 4). For completeness, we include the message complexity and communication complexity for both the standard point-to-point model as well as the "broadcast" model discussed above. As for computational costs, the protocol of [16] requires some players to compute $O(n)$ "full" modular exponentiations (in addition to $O(1)$ signatures and $O(1)$ signature verifications); the protocol of [13] requires the initiator of the protocol to perform $O(n)$ public-key encryptions and one signature computation; and our protocol requires each player to compute $O(n \log n)$ modular multiplications and to verify $O(n)$ signatures (in addition to $O(1)$ signature computations and 3 "full" modular exponentiations).

1.4 Outline

In Section 2, we review the security model of Bresson, et al. [16]. We present our compiler in Section 3 and a two-round protocol secure against passive adversaries in Section 4. Applying our compiler to this protocol gives our main result: an efficient, fully-scalable, and constant-round group AKE protocol.

2 The Model and Preliminaries

Our security model is the standard one of Bresson, et al. [16] which builds on prior work from the 2-party setting [9, 10, 7] and which has been widely used to analyze group key-exchange protocols (e.g., [14, 15, 13]). We explicitly define notions of security for both passive and active adversaries; this will be necessary for stating and proving meaningful results about our compiler in Section 3.

Participants and initialization. We assume for simplicity a fixed, polynomial-size set $\mathcal{P} = \{U_1, \ldots, U_\ell\}$ of potential participants. Any subset of $\mathcal{P}$ may decide at any point to establish a session key, and we do not assume that these subsets are always the same size or always include the same participants. Before the protocol is run for the first time, an initialization phase occurs during which each participant $U \in \mathcal{P}$ runs an algorithm $\mathcal{G}(1^k)$ to generate public/private keys (PK_U, SK_U). Each player U stores SK_U, and the vector $\langle PK_i \rangle_{1 \leq i \leq |\mathcal{P}|}$ is known by all participants (and is also known by the adversary).

Adversarial model. In the real world, a protocol determines how principals behave in response to signals from their environment. In the model, these signals are sent by the adversary. Each principal can execute the protocol multiple times with different partners; this is modeled by allowing each principal an unlimited number of *instances* with which to execute the protocol. We denote instance i of user U as Π_U^i. A given instance may be used only once. Each instance Π_U^i has associated with it the variables state_U^i, term_U^i, acc_U^i, used_U^i, sid_U^i, pid_U^i, and sk_U^i; the last of these is the *session key* whose computation is the goal of the protocol, while the function of the remaining variables is as in [7].

The adversary is assumed to have complete control over all communication in the network. An adversary's interaction with the principals in the network (more specifically, with the various instances) is modeled by the following *oracles*:

- $\mathsf{Send}(U, i, M)$ — This sends message M to instance Π_U^i, and outputs the reply generated by this instance. We allow the adversary to prompt the unused instance Π_U^i to initiate the protocol with partners $U_2, \ldots, U_n$ by calling $\mathsf{Send}(U, i, \langle U_2, \ldots, U_n \rangle)$.

- $\mathsf{Execute}(U_1, \ldots, U_n)$ — This executes the protocol between unused instances of players $U_1, \ldots, U_n \in \mathcal{P}$ and outputs the transcript of the execution. The number of group members and their identities are chosen by the adversary.

- $\mathsf{Reveal}(U, i)$ — This outputs session key sk_U^i.

- $\mathsf{Corrupt}(U)$ — This outputs the long-term secret key SK_U of player U.

- $\mathsf{Test}(U, i)$ — This query is allowed only once, at any time during the adversary's execution. A random bit b is generated; if $b = 1$ the adversary is given sk_U^i, and if $b = 0$ the adversary is given a random session key.

A *passive adversary* is given access to the Execute, Reveal, Corrupt, and Test oracles, while an *active adversary* is additionally given access to the Send oracle. (Here, even though the Execute oracle can be simulated via repeated calls to the Send oracle, the presence of the Execute oracle allows for a tighter definition of forward secrecy as well as a more exact concrete security analysis.)

Partnering. Partnering is defined via *session IDs* and *partner IDs*. The session ID for instance Π_U^i (denoted sid_U^i) is a protocol-specified function of all communication sent and received by Π_U^i; for our purposes, we will simply set sid_U^i equal to the concatenation of all messages sent and received by Π_U^i during the course of its execution. The partner ID for instance Π_U^i (denoted pid_U^i) consists of the identities of the players in the group with whom Π_U^i intends to establish a session key, including U itself; note that these identities are always clear from the initial call to the Send or Execute oracles. We say instances Π_U^i and $\Pi_{U'}^j$ are *partnered* iff (1) $\mathsf{pid}_U^i = \mathsf{pid}_{U'}^j$ and (2) $\mathsf{sid}_U^i = \mathsf{sid}_{U'}^j$. Our definition of partnering is much simpler than that of [16] since, in our protocols, all messages are sent to all other members of the group taking part in the protocol.

Correctness. Of course, we wish to rule out "useless" protocols from consideration. In the standard way, we require that for all U, U', i, j such that $\mathsf{sid}_U^i = \mathsf{sid}_{U'}^j$, $\mathsf{pid}_U^i = \mathsf{pid}_{U'}^j$, and $\mathsf{acc}_U^i = \mathsf{acc}_{U'}^j = \text{TRUE}$ it is the case that $\mathsf{sk}_U^i = \mathsf{sk}_{U'}^j \neq \text{NULL}$.

Freshness. Following [7, 16, 28], we define a notion of *freshness* appropriate for the goal of forward secrecy. An instance Π_U^i is *fresh* unless one of the following is true (in the following, $\Pi_{U'}^j$ is any instance partnered with Π_U^i): (1) at some point, the adversary queried $\mathsf{Reveal}(U, i)$ or $\mathsf{Reveal}(U', j)$; or (2) a query $\mathsf{Corrupt}(V)$ (with $V \in \mathsf{pid}_U^i$) was asked before a query of the form $\mathsf{Send}(U, i, *)$ or[4] $\mathsf{Send}(U', j, *)$.

Definitions of security. We say event Succ occurs if the adversary queries the Test oracle on a fresh instance Π_U^i for which $\mathsf{acc}_U^i = \text{TRUE}$ and correctly guesses the bit b used by the Test oracle in answering this query. The advantage of an adversary $\mathcal{A}$ in attacking protocol P is defined as $\mathsf{Adv}_{\mathcal{A},P}(k) \stackrel{\text{def}}{=} |2 \cdot \Pr[\mathsf{Succ}] - 1|$. We say protocol P is a *secure group key exchange (KE) protocol* if it is secure against a passive adversary; that is, for any PPT passive adversary $\mathcal{A}$ it is the case that $\mathsf{Adv}_{\mathcal{A},P}(k)$ is negligible. We say protocol P is a *secure authenticated group key exchange (AKE) protocol* if it is secure against an active adversary; that is, for any PPT active adversary $\mathcal{A}$ it is the case that $\mathsf{Adv}_{\mathcal{A},P}(k)$ is negligible.

To enable a concrete security analysis, we define $\mathsf{Adv}_P^{\mathsf{KE-fs}}(t, q_{\text{ex}})$ to be the maximum advantage of any passive adversary attacking P, running in time t and making q_{ex} calls to the $\mathsf{Execute}$ oracle. Similarly, we define $\mathsf{Adv}_P^{\mathsf{AKE-fs}}(t, q_{\text{ex}}, q_{\text{s}})$ to be the maximum advantage of any active adversary attacking P, running in time t and making q_{ex} calls to the $\mathsf{Execute}$ oracle and q_{s} calls to the Send oracle.

Protocols without forward secrecy. Throughout this paper we will be concerned primarily with protocols achieving forward secrecy; the definitions above already incorporate this requirement since the adversary has access to the $\mathsf{Corrupt}$ oracle in each case. However, our compiler may also be applied to KE protocols which do not achieve forward secrecy (cf. Theorem 1). For completeness, we define $\mathsf{Adv}_P^{\mathsf{KE}}(t, q_{\text{ex}})$ and $\mathsf{Adv}_P^{\mathsf{AKE}}(t, q_{\text{ex}}, q_{\text{s}})$ in a manner completely analogous to the above, with the exception that the adversary in each case no longer has access to the $\mathsf{Corrupt}$ oracle.

Authentication. We do not define any notion of explicit authentication or, equivalently, confirmation that the other members of the group have computed the common key. Indeed, our protocols do not explicitly provide such confirmation. However, explicit authentication in our protocols can be achieved at little additional cost. Previous work (e.g., [16, Sec. 7]) shows how to achieve explicit authentication for any secure group AKE protocol using one additional round and minimal extra computation. (Although [16] use the random oracle model, their techniques can be extended to the standard model by replacing the random oracle with a pseudorandom function.) Applying their transformation to our final protocol results in a constant-round group AKE protocol with explicit authentication.

[4] While not necessary in general, the second part of this requirement is used in the proof of Theorem 2. This requirement is anyway quite natural — if an adversary corrupts a user $V \in \mathsf{pid}_U^i$, it likely obtains any current session keys (in particular, the one shared with U) in addition to V's long-term secret key.

2.1 Notes on the Definition

Although the above definition is standard for the analysis of group key-exchange protocols — it is the definition used, e.g., in [16, 14, 15, 13] — there are a number of concerns that it does *not* address. For one, it does not offer any protection against malicious insiders or users who do not honestly follow the protocol. The definition is also not intended to ensure any form of "agreement" (in the sense of [25]); in fact, since the model gives the adversary *complete* control over all communication in the network (i.e., in addition to delaying messages, the adversary may modify messages or refuse to deliver them at all!) and allows the adversary to corrupt *all* parties, full-fledged agreement is clearly impossible. Finally, the definition inherently does not protect against "denial of service" attacks, and cannot prevent the adversary from causing an honest instance to "hang" indefinitely; this is a consequence of the adversary's ability to refuse to deliver messages.

Some of these concerns can be addressed — at least partially — within the model above. For example, it seems that the following approach can be used following *any* group AKE protocol to achieve confirmation that all (non-corrupted) participants have computed a matching session key: after computing key sk, each player U_i computes $x_i = H(\mathsf{sk}\|U_i)$, signs x_i, broadcasts x_i and the corresponding signature, and computes the "actual" session key $\mathsf{sk}' = H(\mathsf{sk}\|\bot)$ (here, H is modeled as a random oracle and "$\bot$" represents some distinguished string); other players check the validity of the broadcast values in the obvious way.[5] Although this does not provide agreement (since an adversary can still refuse to deliver messages to some of the participants), it *does* ensure the equality of any keys generated by partnered instances.

Rigorously proving the above approach secure, as well as addressing the other concerns mentioned above (by necessary modifications of the adversarial model), represents an interesting direction for future research.

3 A Scalable Compiler for Group AKE Protocols

3.1 Description of the Compiler

We show here a compiler transforming any secure group KE protocol P to a secure group AKE protocol P' (recall from Section 2 that a group *KE* protocol protects against a passive adversary only, while a group *AKE* protocol additionally protects against an active adversary). Without loss of generality, we assume the following about the original protocol P: (1) Each message sent by an instance Π_U^i during execution of P includes the sender's identity U as well as a sequence number which begins at 1 and is incremented each time Π_U^i sends a message (in other words, the j^{th} message sent by an instance Π_U^i has the form $U|j|m$); (2) every message of the protocol is sent — via point-to-point links — to every member of the group taking part in the execution of the protocol (that is, Π_U^i sends each message to all users in pid_U^i). For simplicity, we refer to this as "broadcasting a message" but stress that we do *not* assume a broadcast channel and, in particular, an active adversary can deliver different messages to different members of the group or refuse to deliver messages to some of the participants. Note that any secure group KE protocol $\widetilde{P}$ can be readily converted to

[5] This differs from the approach of [16, Sec. 7] in that we require a signature on the broadcast value x_i.

a secure group KE protocol P in which the above assumptions hold (security of P follows trivially from the security of $\widetilde{P}$ since here only a passive adversary is assumed); furthermore, the complexities of P and $\widetilde{P}$ are essentially the same in all respects.

Let $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be a signature scheme which is strongly unforgeable under adaptive chosen message attack (where "strong" means that an adversary is also unable to forge a new signature for a previously-signed message), and let $\mathsf{Succ}_\Sigma(t)$ denote the maximum advantage of any adversary running in time t in forging a new message/signature pair. For simplicity, we assume that the signature length is independent of the length of the message signed; this is easy to achieve in practice by hashing the message (using a collision-resistant hash function) before signing. Given protocol P as above, our compiler constructs a new protocol P' as follows:

1. During the initialization phase, each party $U \in \mathcal{P}$ generates the verification/signing keys (PK'_U, SK'_U) by running $\mathsf{Gen}(1^k)$. This is in addition to any keys (PK_U, SK_U) needed as part of the initialization phase for P.

2. Let $U_1, \ldots, U_n$ be the identities (in lexicographic order) of users wishing to establish a common key, and let $\mathcal{U} = U_1|\cdots|U_n$. Each user U_i begins by choosing a random nonce $r_i \in \{0,1\}^k$ and broadcasting $U_i|0|r_i$ (note we assign this message the sequence number "0"). After receiving the initial broadcast message from all other parties, each instance Π_U^j (for $U \in \mathcal{U}$) sets $\mathsf{nonces}_U^j = ((U_1, r_1), \ldots, (U_n, r_n))$ and stores this as part of its state information.

3. The members of the group now execute P with the following changes:

   - Whenever instance Π_U^i is supposed to broadcast $U|j|m$ as part of protocol P, the instance computes $\sigma \leftarrow \mathsf{Sign}_{SK'_U}(j|m|\mathsf{nonces}_U^i)$ and then broadcasts $U|j|m|\sigma$.

   - When instance Π_U^i receives message $V|j|m|\sigma$, it checks that: (1) $V \in \mathsf{pid}_U^i$, (2) j is the next expected sequence number for messages from V, and, finally, (3) $\mathsf{Vrfy}_{PK'_V}(j|m|\mathsf{nonces}_U^i, \sigma) = 1$. If any of these are untrue, Π_U^i aborts the protocol and sets $\mathsf{acc}_U^i = \text{FALSE}$ and $\mathsf{sk}_U^i = \text{NULL}$. Otherwise, Π_U^i continues as it would in P upon receiving message $V|j|m$.

4. Each non-aborted instance computes the session key as in P.

3.2 Compiling Protocols without Forward Secrecy

This compiler of the previous section may be applied to any group KE protocol P, regardless of whether P achieves forward secrecy. Clearly, we cannot expect the compiled protocol to achieve forward secrecy if the original protocol does not. In this case we simply show that the compiled protocol is an *authenticated* key-exchange protocol; i.e., it is secure against an active adversary.

Theorem 1 *If P is a secure group KE protocol (without forward secrecy), then P' given by the above compiler is a secure group AKE protocol (without forward secrecy). Namely:*

$$\mathsf{Adv}_{P'}^{\mathsf{AKE}}(t, q_{\text{ex}}, q_{\text{s}}) \leq \mathsf{Adv}_P^{\mathsf{KE}}(t', q_{\text{ex}} + \frac{q_{\text{s}}}{2}) + |\mathcal{P}| \cdot \mathsf{Succ}_\Sigma(t') + \frac{q_{\text{s}}^2 + q_{\text{ex}} q_{\text{s}}}{2^k},$$

where $t' = t + (|\mathcal{P}|q_{\rm ex} + q_{\rm s}) \cdot t_{P'}$ and $t_{P'}$ is the time required for execution of P' by any party.

Proof Given an *active* adversary $\mathcal{A}'$ attacking P', we will construct a *passive* adversary $\mathcal{A}$ attacking P; relating the success probabilities of $\mathcal{A}'$ and $\mathcal{A}$ gives the stated result. Note that neither $\mathcal{A}'$ nor $\mathcal{A}$ is given access to the Corrupt oracle since neither P' nor P is intended to provide forward secrecy (cf. Section 2); indeed, it is this feature which leads to a simpler proof than in the case of Theorem 2 where forward secrecy will be taken into account.

Before describing $\mathcal{A}$, we first define events Forge and Repeat and bound their respective probabilities. Let Forge be the event that $\mathcal{A}'$ outputs a new, valid message/signature pair with respect to the public key PK'_U of some user $U \in \mathcal{P}$ at any point during its execution, and let $\Pr[\mathsf{Forge}]$ denote $\Pr_{\mathcal{A}',P'}[\mathsf{Forge}]$ for brevity. (Formally, Forge is the event that $\mathcal{A}'$ makes a query of the form $\mathsf{Send}(V, i, U|j|m|\sigma)$ where $\mathsf{Vrfy}_{PK'_U}(j|m|\mathsf{nonces}^i_V, \sigma) = 1$ and σ was not previously output by any instance of player U as a signature on $j|m|\mathsf{nonces}^i_V$. With this in mind, however, we use the more informal description of Forge for the remainder of this paragraph.) Using $\mathcal{A}'$, we may construct an algorithm $\mathcal{F}$ that forges a signature with respect to signature scheme Σ as follows: given a public key PK, algorithm $\mathcal{F}$ chooses a random $U \in \mathcal{P}$, sets $PK'_U = PK$, and honestly generates all other public/private keys for the system (this includes both the keys $\{PK'_{\tilde{U}}, SK'_{\tilde{U}}\}_{\tilde{U} \neq U}$ used by the compiler as well as all keys $\{PK_U, SK_U\}_{U \in \mathcal{P}}$, if any, needed for execution of the original protocol P). The forger $\mathcal{F}$ simulates all oracle queries of $\mathcal{A}'$ in the natural way by executing protocol P' itself, obtaining the necessary signatures with respect to PK'_U, as needed, from its signing oracle. In this way, $\mathcal{F}$ provides a perfect simulation for $\mathcal{A}'$. Now, if $\mathcal{A}'$ ever outputs a new, valid message/signature pair with respect to $PK'_U = PK$, then $\mathcal{F}$ outputs this pair as its forgery (note that $\mathcal{F}$ can efficiently determine whether a forgery has occurred). The success probability of $\mathcal{F}$ is exactly $\frac{\Pr[\mathsf{Forge}]}{|\mathcal{P}|}$; this immediately implies that

$$\Pr[\mathsf{Forge}] \leq |\mathcal{P}| \cdot \mathsf{Succ}_{\Sigma}(t').$$

Let Repeat be the event that a nonce used by any user in response to a Send query was used previously by that user (in response to either an Execute *or* a Send query). It is immediate that

$$\Pr[\mathsf{Repeat}] \leq \frac{q_s q_e + q_s^2}{2^k}.$$

Having bounded the probabilities of events Forge and Repeat, we now describe the construction of the passive adversary $\mathcal{A}$ attacking protocol P. At a high level, $\mathcal{A}$ simply runs $\mathcal{A}'$ as a subroutine and simulates the oracle queries of $\mathcal{A}'$ using its own queries to the Execute oracle. If Forge or Repeat occur, $\mathcal{A}$ aborts and outputs a random bit; otherwise, it outputs whatever bit is eventually output by $\mathcal{A}'$. We now describe the simulation in more detail. Recall that as part of the initial setup, $\mathcal{A}$ is given public keys $\{PK_U\}_{U \in \mathcal{P}}$ if any are defined as part of protocol P. We first have $\mathcal{A}$ run $\mathsf{Gen}(1^k)$ to generate keys $\{PK'_U, SK'_U\}_{U \in \mathcal{P}}$; the set of public keys $\{PK'_U, PK_U\}_{U \in \mathcal{P}}$ is then given to $\mathcal{A}'$. Then $\mathcal{A}$ runs $\mathcal{A}'$, simulating the oracle queries of $\mathcal{A}'$ as follows (to aid the simulation, $\mathcal{A}$ maintains a list Nonces whose function will become clear from the description):

Execute queries. Assume $\mathcal{A}'$ makes a query $\mathsf{Execute}(U_1, \ldots, U_n)$, and let $\mathcal{U} \stackrel{\rm def}{=} U_1 | \cdots | U_n$ (where these are assumed to be in lexicographic order). Adversary $\mathcal{A}$ sends the same query to its $\mathsf{Execute}$ oracle and receives in return a transcript T of an execution of P. Next, $\mathcal{A}$ chooses random $r_1, \ldots, r_n \in \{0,1\}^k$, sets $\mathsf{nonces} = ((U_1, r_1), \ldots, (U_n, r_n))$, and stores (nonces, T) in Nonces. To simulate a transcript T' of an execution of P', $\mathcal{A}$ sets the initial messages of T' to $\{U_i|0|r_i\}_{1 \le i \le n}$. Furthermore, for each message $U|j|m$ in transcript T, algorithm $\mathcal{A}$ computes signature $\sigma \leftarrow \mathsf{Sign}_{SK'_U}(j|m|\mathsf{nonces})$ and places $U|j|m|\sigma$ in T'. When done, the complete transcript T' is given to $\mathcal{A}'$.

Send queries. Denote the initial Send query to an instance (denoting a request for protocol initiation) by Send_0; note that for any particular instance Π_U^ℓ this query always has the form $\mathsf{Send}_0(U, \ell, \langle U_1, \ldots, U_n \rangle)$. Denote the second Send query to an instance by Send_1; for an instance Π_U^ℓ as before, we may assume without loss of generality that this query has the form $\mathsf{Send}_1(U, \ell, \langle U_1|0|r_1, \ldots, U_n|0|r_n \rangle)$ with $r_i \in \{0,1\}^k$ for all i. Following a Send_1 query to instance Π_U^ℓ, define nonces_U^ℓ as the lexicographic ordering (according to the U's) of $((U, r_U^\ell), (U_1, r_1), \ldots, (U_n, r_n))$, where r_U^ℓ is the nonce generated by Π_U^ℓ.

In response to any Send query, $\mathcal{A}$ proceeds as follows:

- On query $\mathsf{Send}_0(U, \ell, *)$, $\mathcal{A}$ chooses random $r_U^\ell \in \{0,1\}^k$ and replies to $\mathcal{A}'$ with $U|0|r_U^\ell$.
- On a Send_1 query to an instance Π_U^ℓ, the values of pid_U^ℓ and nonces_U^ℓ are defined (by the previous Send_0 query to this instance and the current query, respectively). $\mathcal{A}$ looks in Nonces for an entry of the form $(\mathsf{nonces}_U^\ell, T)$; if no entry of this form exists, then $\mathcal{A}$ queries $\mathsf{Execute}(\mathsf{pid}_U^\ell)$, receives in return a transcript T, and stores $(\mathsf{nonces}_U^\ell, T)$ in Nonces. In either case, $\mathcal{A}$ now has a transcript T associated with the value nonces_U^ℓ. Next, $\mathcal{A}$ finds the (unique) message of the form $U|1|m$ in T, computes the signature $\sigma \leftarrow \mathsf{Sign}_{SK'_U}(1|m|\mathsf{nonces}_U^\ell)$, and returns $U|1|m|\sigma$ to $\mathcal{A}'$.
- In response to any other Send query to an instance Π_U^ℓ, $\mathcal{A}$ first verifies correctness of the current incoming message(s) as in the specification of the compiler and terminates the instance if verification fails. Assuming verification succeeds, $\mathcal{A}$ finds an entry $(\mathsf{nonces}_U^\ell, T)$ in Nonces (such an entry must exist if we assume, without loss of generality, that $\mathcal{A}'$ had previously issued a Send_1 query to Π_U^ℓ), locates the appropriate message $U|j|m$ in transcript T, computes the signature $\sigma \leftarrow \mathsf{Sign}_{SK'_U}(j|m|\mathsf{nonces}_U^\ell)$, and replies to $\mathcal{A}'$ with $U|j|m|\sigma$.

Reveal/Test queries. When $\mathcal{A}'$ queries $\mathsf{Reveal}(U, \ell)$ or $\mathsf{Test}(U, \ell)$ for an instance for which $\mathsf{acc}_U^\ell = \text{TRUE}$, it must be the case that $T' \stackrel{\text{def}}{=} \mathsf{sid}_U^\ell$ is defined (recall that sid_U^ℓ is simply the transcript of the execution for instance Π_U^ℓ). Let T denote the underlying transcript of protocol P that is contained within transcript T', obtained by simply "stripping" the signatures from T'. Now, $\mathcal{A}$ finds any query it made to its own $\mathsf{Execute}$ oracle which resulted in transcript T (assuming events Repeat and Forge do not occur, at least one such query will exist), makes the appropriate Reveal or Test query to one of the instances involved in this query (it does not matter which, since they are all partnered and all hold the same session key), and returns the result to $\mathcal{A}'$.

We claim that the above is a perfect simulation for $\mathcal{A}'$ as long as Forge and Repeat do not occur. First, note that the $\mathsf{Execute}$ queries of $\mathcal{A}'$, as well as all $\mathsf{Reveal}/\mathsf{Test}$ queries made by $\mathcal{A}'$ to instances that $\mathcal{A}'$ initiated via an $\mathsf{Execute}$ query, are simulated perfectly. As for the Send queries of $\mathcal{A}'$, we may note the following: Send_0 queries are simulated perfectly. Send_1 queries are simulated perfectly assuming event Repeat does not occur. All other Send queries, as well as all $\mathsf{Reveal}/\mathsf{Test}$ queries made to instances initiated via Send queries, are simulated perfectly assuming that neither Repeat nor Forge occur.

Letting $\mathsf{Bad} \stackrel{\text{def}}{=} \mathsf{Forge} \vee \mathsf{Repeat}$, a straightforward calculation shows:

$$\begin{aligned}
\mathsf{Adv}_{\mathcal{A},P} &\stackrel{\text{def}}{=} 2 \cdot \left|\Pr_{\mathcal{A},P}[\mathsf{Succc}] - \tfrac{1}{2}\right| \\
&= 2 \cdot \left|\Pr_{\mathcal{A}',P'}[\mathsf{Succ} \wedge \overline{\mathsf{Bad}}] + \tfrac{1}{2}\Pr_{\mathcal{A}',P'}[\mathsf{Bad}] - \tfrac{1}{2}\right| \\
&= 2 \cdot \left|\Pr_{\mathcal{A}',P'}[\mathsf{Succ}] - \Pr_{\mathcal{A}',P'}[\mathsf{Succ} \wedge \mathsf{Bad}] + \tfrac{1}{2}\Pr_{\mathcal{A}',P'}[\mathsf{Bad}] - \tfrac{1}{2}\right| \\
&\geq \left|2 \cdot \Pr_{\mathcal{A}',P'}[\mathsf{Succ}] - 1\right| - \left|\Pr_{\mathcal{A}',P'}[\mathsf{Bad}] - 2 \cdot \Pr_{\mathcal{A}',P'}[\mathsf{Succ} \wedge \mathsf{Bad}]\right| \\
&\geq \mathsf{Adv}_{\mathcal{A}',P'} - \Pr_{\mathcal{A}',P'}[\mathsf{Bad}].
\end{aligned}$$

Since $\mathsf{Adv}_{\mathcal{A},P} \leq \mathsf{Adv}_P^{\mathsf{KE}}(t', q_{\text{ex}} + q_{\text{s}}/2)$ by assumption (note that $\mathcal{A}$ makes at most $q_{\text{ex}} + q_{\text{s}}/2$ queries to its $\mathsf{Execute}$ oracle), we obtain:

$$\mathsf{Adv}_{P'}^{\mathsf{AKE}} \leq \mathsf{Adv}_P^{\mathsf{KE}}(t', q_{\text{ex}} + \frac{q_{\text{s}}}{2}) + \Pr[\mathsf{Forge}] + \Pr[\mathsf{Repeat}],$$

yielding the statement of the theorem. ■

3.3 Compiling Protocols which Achieve Forward Secrecy

As noted earlier, our compiler may also be applied to key-exchange protocols which *do* achieve forward secrecy; in this case, the compiled protocol does too. Although we consider this the more interesting result, we have chosen to present the proof of Theorem 1 first because the former proof is simpler yet contains the essential ideas used here.

Before giving the formal proof, we begin with a high-level overview of how the proof differs in this case. As in the proof of Theorem 1, we will transform an active adversary $\mathcal{A}'$ attacking protocol P' into a passive adversary $\mathcal{A}$ attacking protocol P. In the proof of Theorem 1, adversary $\mathcal{A}$ queried its $\mathsf{Execute}$ oracle (to obtain a transcript T) for each $\mathsf{Execute}$ query of $\mathcal{A}'$ and also for each query $\mathsf{Send}_1(U, \ell, *)$ of $\mathcal{A}'$ which resulted in a new value of nonces_U^ℓ; this transcript is then "patched" (by generating appropriate signatures) to give a transcript T' which is then used to answer queries of $\mathcal{A}'$. An essential reason this yields a good simulation is that $\mathcal{A}'$ is "limited" to sending messages already contained in T'; this is because $\mathcal{A}'$ cannot forge signatures with respect to any of the users, and because nonces do not repeat (with high probability).

Here, however, we cannot quite follow the same approach; in fact, doing so leads to a potential problem in the simulation. Namely, since $\mathcal{A}'$ can now ask a $\mathsf{Corrupt}$ query to obtain secret keys, it is possible for $\mathcal{A}'$ to send messages that are *not* contained in the "patched" transcript T'. (That is, $\mathcal{A}'$ might query $\mathsf{Send}_1(U, \ell, *)$ and then, at some later point in time — but before instance Π_U^ℓ terminates — corrupt a player $V \in \mathsf{pid}_U^\ell$. This will allow $\mathcal{A}'$ to sign messages on behalf of V and therefore to send arbitrary messages of its choice to Π_U^ℓ.) In this case, $\mathcal{A}$ must be able to simulate the actions of Π_U^ℓ without recourse to a fixed transcript T obtained from its $\mathsf{Execute}$ oracle.

To avoid this potential problem with the simulation, we adopt the following modified approach (described informally here): for all but at most one Send query of $\mathcal{A}'$, algorithm $\mathcal{A}$

will simulate the Send queries *itself* by running the protocol on its own (rather than via calls to its own $\mathsf{Execute}$ oracle). A key point is that $\mathcal{A}$ can efficiently perform this simulation by making the necessary $\mathsf{Corrupt}$ queries to obtain any private information needed to execute the underlying protocol. For the (at most one) Send query that $\mathcal{A}$ does not simulate on its own, it will use its $\mathsf{Execute}$ oracle to obtain a transcript of P and simulate an execution of P' as in the proof of Theorem 1. In essence, $\mathcal{A}$ is guessing that $\mathcal{A}'$ will eventually issue a Test query for this instance, which degrades the concrete security by a factor of roughly q_{s}. The formal proof is slightly more complex (in particular, $\mathcal{A}$ must maintain consistency among the various oracle calls of $\mathcal{A}'$, and we have not discussed how to deal with the $\mathsf{Execute}$ calls of $\mathcal{A}'$), and follows now.

Theorem 2 *If P is a secure group KE protocol achieving forward secrecy, then P' given by the above compiler is a secure group AKE protocol achieving forward secrecy. Namely:*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{AKE-fs}}_{P'}(t, q_{\mathrm{ex}}, q_{\mathrm{s}}) \quad &\leq \quad \frac{q_{\mathrm{s}}}{2} \cdot \mathsf{Adv}^{\mathsf{KE-fs}}_{P}(t', 1) + \mathsf{Adv}^{\mathsf{KE-fs}}_{P}(t', q_{\mathrm{ex}}) \\ &\qquad + |\mathcal{P}| \cdot \mathsf{Succ}_{\Sigma}(t') + \frac{q_{\mathrm{s}}^2 + q_{\mathrm{ex}} q_{\mathrm{s}}}{2^k},\end{aligned}$$

where t' is as in Theorem 1.

Proof Define event Repeat exactly as in the proof of Theorem 1. Event Forge is defined slightly differently; namely, Forge is now the event that $\mathcal{A}'$ outputs a new, valid message/signature pair with respect to the public key PK'_U of some user $U \in \mathcal{P}$ *before* querying $\mathsf{Corrupt}(U)$. The bounds on the probabilities of these events, however, are exactly as in the proof of Theorem 1.

We condition the success of $\mathcal{A}'$ on whether it asks its Test query to an instance which was initialized via an $\mathsf{Execute}$ query or via a Send query. More formally, let Ex be the event that $\mathcal{A}'$ makes its query $\mathsf{Test}(U, i)$ to an instance Π^i_U such that $\mathcal{A}'$ never made a query of the form $\mathsf{Send}(U, i, *)$ (and therefore *did* make a query $\mathsf{Execute}(U, i, *)$). We also define $\mathsf{Se} \stackrel{\text{def}}{=} \overline{\mathsf{Ex}}$. We now bound the probabilities of $\Pr_{\mathcal{A}',P'}[\mathsf{Succ} \wedge \mathsf{Ex}]$ and $\Pr_{\mathcal{A}',P'}[\mathsf{Succ} \wedge \mathsf{Se}]$ by constructing appropriate adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$.

The initial behavior of adversaries $\mathcal{A}_1/\mathcal{A}_2$ is the same, and we therefore describe it now once and for all. Recall that as part of the initial setup, adversary $\mathcal{A}_1/\mathcal{A}_2$ is given public keys $\{PK_U\}_{U\in\mathcal{P}}$ if any are defined as part of protocol P. First, $\mathcal{A}_1/\mathcal{A}_2$ obtains all secret keys $\{SK_U\}_{U\in\mathcal{P}}$ using multiple $\mathsf{Corrupt}$ queries. Next, $\mathcal{A}_1/\mathcal{A}_2$ runs $\mathsf{Gen}(1^k)$ to generate keys (PK'_U, SK'_U) for each $U \in \mathcal{P}$. Finally, the set of public keys $\{PK'_U, PK_U\}_{U\in\mathcal{P}}$ is given to $\mathcal{A}'$. We stress that $\mathcal{A}_1/\mathcal{A}_2$ now has the secret information of all parties in the network; yet, since P is supposed to provide forward secrecy, this should not affect the security of the protocol against a passive attack (as will be mounted by $\mathcal{A}_1/\mathcal{A}_2$). We now have $\mathcal{A}_1/\mathcal{A}_2$ run $\mathcal{A}'$, simulating the oracle queries of $\mathcal{A}'$ as described below.

We first describe the simulation provided by $\mathcal{A}_1$ (this description is informal, since it is similar, though not identical, to the adversary $\mathcal{A}$ constructed in the proof of Theorem 1). $\mathcal{A}_1$ simulates all $\mathsf{Execute}$ queries of $\mathcal{A}'$ via $\mathsf{Execute}$ queries of its own; namely, by forwarding the query of $\mathcal{A}'$ to its $\mathsf{Execute}$ oracle to obtain transcript T, "patching" the transcript by generating the appropriate signatures as in the proof of Theorem 1, and giving the modified transcript T' to $\mathcal{A}'$. All Send queries of $\mathcal{A}'$, however, are answered by having $\mathcal{A}_1$ execute protocol P' *itself* (without making any calls to its $\mathsf{Execute}$ oracle); note that $\mathcal{A}_1$ can do this since it has the secret information of all parties. Reveal queries to instances initiated by $\mathcal{A}'$ via $\mathsf{Execute}$ queries are answered by having $\mathcal{A}_1$ make the appropriate Reveal query, as in the proof of Theorem 1; for Reveal queries to instances initiated via Send queries, $\mathcal{A}_1$ simply computes and returns the correct session key itself. $\mathsf{Corrupt}$ queries are answered in the obvious way. Finally, when $\mathcal{A}'$ makes its query $\mathsf{Test}(U, i)$, $\mathcal{A}_1$ checks whether $\mathcal{A}'$ has ever made a query of the form $\mathsf{Send}(U, i, *)$ (i.e., whether event Se has occurred). If so, $\mathcal{A}_1$ aborts and outputs a random bit. Otherwise (if event Ex has occurred), $\mathcal{A}_1$ issues the appropriate Test query to its own oracle, returns the result to $\mathcal{A}'$, and outputs whatever $\mathcal{A}'$ outputs.

Note that the simulation provided by $\mathcal{A}_1$ is perfect (even if Repeat or Forge occur) unless $\mathcal{A}_1$ aborts due to the occurrence of event Se. Thus:

$$\Pr_{\mathcal{A}_1,P}[\mathsf{Succc}] = \Pr_{\mathcal{A}',P'}[\mathsf{Succ} \wedge \mathsf{Ex}] + \tfrac{1}{2} \cdot \Pr_{\mathcal{A}',P'}[\mathsf{Se}]. \tag{1}$$

Furthermore, $|2 \cdot \Pr_{\mathcal{A}_1,P}[\mathsf{Succ}] - 1| \leq \mathsf{Adv}_P^{\mathsf{KE-fs}}(t', q_{\rm ex})$ because $\mathcal{A}_1$ makes at most $q_{\rm ex}$ queries to its $\mathsf{Execute}$ oracle.

We now describe $\mathcal{A}_2$. Informally, we have $\mathcal{A}_2$ guess in advance the *first* Send_1 query corresponding to the eventual Test query of $\mathcal{A}'$. This Send query to some instance Π_U^i (and all Send queries to instances partnered with Π_U^i) will be simulated as in the proof of Theorem 1; namely, $\mathcal{A}_2$ will make the appropriate query to its $\mathsf{Execute}$ oracle to obtain transcript T, "patch" T (by generating appropriate signatures) to obtain the modified transcript T', and then use T' to answer queries of $\mathcal{A}'$. All other Send queries of $\mathcal{A}'$ will be answered by having $\mathcal{A}_2$ simulate execution of P' itself. If the guess of $\mathcal{A}_2$ is incorrect, it aborts and outputs a random bit; otherwise, it makes the appropriate Test query, forwards the result to $\mathcal{A}'$, and outputs whatever $\mathcal{A}'$ outputs. A key point here is that if the guess of $\mathcal{A}_2$ is correct then $\mathcal{A}'$ is (informally) "limited" to sending to Π_U^i (and partnered instances) messages contained in T'; adversary $\mathcal{A}'$ can do otherwise only if it queries $\mathsf{Corrupt}(V)$ for some $V \in \mathsf{pid}_U^i$, but then Π_U^i is no longer fresh (and we assume without loss of generality that $\mathcal{A}'$ only makes a Test query to a fresh instance, since it cannot succeed if this is not the case).

We now provide the details. First, $\mathcal{A}_2$ chooses a random $\alpha \in \{1, \ldots, q_{\rm s}/2\}$ (note that $q_{\rm s}/2$ is an upper bound on the number of Send_1 queries made by $\mathcal{A}'$, where a Send_1 query is defined as in the proof of Theorem 1). It then simulates the oracle calls of $\mathcal{A}'$ as described below (to aid the simulation, $\mathcal{A}_2$ maintains a list Nonces whose function will become clear). As in the proof of Theorem 1, $\mathcal{A}_2$ aborts if Repeat or Forge occurs.

Execute queries. All $\mathsf{Execute}$ queries of $\mathcal{A}'$ are answered by having $\mathcal{A}_2$ execute protocol P' itself ($\mathcal{A}_2$ can do this efficiently because it has the secret keys of all players), resulting in a value nonces and a transcript T. $\mathcal{A}_2$ stores $(\mathsf{nonces}, \emptyset)$ in Nonces and returns T to $\mathcal{A}'$.

Send queries.

- On query $\mathsf{Send}_0(U, \ell, *)$ (where a Send_0 query is defined as in the proof of Theorem 1), $\mathcal{A}_2$ chooses random $r_U^\ell \in \{0,1\}^k$ and replies with $U|0|r_U^\ell$.

- On the $\alpha^{\rm th}$ Send_1 query to an instance Π_U^ℓ, the values of pid_U^ℓ and nonces_U^ℓ are defined (by the previous Send_0 query to this instance and the current query, respectively).

$\mathcal{A}_2$ first looks in Nonces for an entry of the form $(\mathsf{nonces}_U^\ell, \emptyset)$; if such an entry exists, $\mathcal{A}_2$ aborts. Also, if $\mathcal{A}'$ had previously asked a $\mathsf{Corrupt}$ query to any user $V \in \mathsf{pid}_U^\ell$, then $\mathcal{A}_2$ aborts. Otherwise, $\mathcal{A}_2$ queries $\mathsf{Execute}(\mathsf{pid}_U^\ell)$, receives in return a transcript T, and stores $(\mathsf{nonces}_U^\ell, T)$ in Nonces. $\mathcal{A}_2$ then finds the message of the form $U|1|m$ in T, computes the signature $\sigma \leftarrow \mathsf{Sign}_{SK'_U}(1|m|\mathsf{nonces}_U^\ell)$, and returns $U|1|m|\sigma$ to $\mathcal{A}'$.

- On any other Send query to an instance Π_U^ℓ, $\mathcal{A}_2$ looks in Nonces for an entry of the form $(\mathsf{nonces}_U^\ell, T)$. If no such entry exists, $\mathcal{A}_2$ stores $(\mathsf{nonces}_U^\ell, \emptyset)$ in Nonces. In this case or if $T = \emptyset$, $\mathcal{A}_2$ responds to this query by executing protocol P' *itself* (i.e., without making any calls to the $\mathsf{Execute}$ oracle); again, $\mathcal{A}_2$ can do this since it has the secret information of all players. On the other hand, if $T \neq \emptyset$ this implies that $\mathcal{A}_2$ has received T in response to its single $\mathsf{Execute}$ query. Now, $\mathcal{A}_2$ verifies correctness of the incoming messages as in the specification of the compiler and terminates the instance if verification fails. Additionally, $\mathcal{A}_2$ aborts if $\mathcal{A}'$ had previously asked a $\mathsf{Corrupt}$ query to any user $V \in \mathsf{pid}_U^\ell$. Otherwise, $\mathcal{A}_2$ locates the appropriate message $U|j|m$ in transcript T, computes the signature $\sigma \leftarrow \mathsf{Sign}_{SK'_U}(j|m|\mathsf{nonces}_U^\ell)$, and replies to $\mathcal{A}'$ with $U|j|m|\sigma$.

Corrupt queries. These are answered in the obvious way.

Reveal queries. When $\mathcal{A}'$ queries $\mathsf{Reveal}(U, \ell)$ for a terminated instance Π_U^ℓ, the value of nonces_U^ℓ is defined. $\mathcal{A}_2$ finds an entry $(\mathsf{nonces}_U^\ell, T)$ in Nonces and aborts if $T \neq \emptyset$. Otherwise, if $T = \emptyset$ it means that $\mathcal{A}_2$ has executed protocol P' itself; $\mathcal{A}_2$ can therefore compute the appropriate session key and return it to $\mathcal{A}'$.

Test query. When $\mathcal{A}'$ queries $\mathsf{Test}(U, i)$ for a terminated instance, $\mathcal{A}_2$ finds an entry $(\mathsf{nonces}_U^\ell, T)$ in Nonces. If $T = \emptyset$, then $\mathcal{A}_2$ aborts. Otherwise, $\mathcal{A}_2$ makes the appropriate Test query (for one of the instances it had activated via its single $\mathsf{Execute}$ query) and returns the result to $\mathcal{A}'$.

Let Bad denote $\mathsf{Forge} \cup \mathsf{Repeat}$, and recall that $\mathcal{A}_2$ aborts immediately if Bad occurs. From the above simulation, we may also note that $\mathcal{A}_2$ aborts if Ex occurs (since in this case $T = \emptyset$ when $\mathcal{A}'$ makes its Test query). On the other hand, if neither Bad nor Ex occur, then $\mathcal{A}_2$ does not abort (i.e., it correctly guesses the value of α) with probability exactly $2/q_{\mathrm{s}}$. Furthermore, the simulation is perfect in case $\mathcal{A}_2$ does not abort. Putting this together (and letting Guess denote the probability that $\mathcal{A}_2$ correctly guesses α) implies

$$\Pr_{\mathcal{A}_2,P}[\mathsf{Succ}] = \tfrac{2}{q_{\mathrm{s}}} \cdot \Pr_{\mathcal{A}',P'}[\mathsf{Succ} \wedge \mathsf{Se} \wedge \overline{\mathsf{Bad}}] + \tfrac{1}{2} \cdot \Pr[\mathsf{Ex} \vee \overline{\mathsf{Guess}} \vee \mathsf{Bad}]. \qquad (2)$$

Furthermore, $|2 \cdot \Pr_{\mathcal{A}_2,P}[\mathsf{Succ}] - 1| \leq \mathsf{Adv}_P^{\mathsf{KE-fs}}(t', 1)$ since $\mathcal{A}_2$ makes only a single query to its $\mathsf{Execute}$ oracle.

Using Eqs. (1) and (2) we obtain (in the following, $\Pr[\cdot]$ always denotes $\Pr_{\mathcal{A}',P'}[\cdot]$):

$$\begin{aligned}
&\left|2 \cdot \Pr_{\mathcal{A}',P'}[\mathsf{Succ}] - 1\right| \\
&\quad = \; 2 \cdot \left|\Pr[\mathsf{Succ} \wedge \mathsf{Ex}] + \Pr[\mathsf{Succ} \wedge \mathsf{Se} \wedge \mathsf{Bad}] + \Pr[\mathsf{Succ} \wedge \mathsf{Se} \wedge \overline{\mathsf{Bad}}] - \tfrac{1}{2}\right| \\
&\quad \leq \; \mathsf{Adv}_P^{\mathsf{KE-fs}}(t', q_{\mathrm{ex}}) + 2 \cdot \left|\Pr[\mathsf{Succ} \wedge \mathsf{Se} \wedge \overline{\mathsf{Bad}}] + \Pr[\mathsf{Succ} \wedge \mathsf{Se} \wedge \mathsf{Bad}] - \tfrac{1}{2}\Pr[\mathsf{Se}]\right| \\
&\quad \leq \; \mathsf{Adv}_P^{\mathsf{KE-fs}}(t', q_{\mathrm{ex}}) + \tfrac{q_{\mathrm{s}}}{2} \cdot \mathsf{Adv}_P^{\mathsf{KE-fs}}(t', 1)
\end{aligned}$$

$$
\begin{aligned}
&\quad + \left|2 \cdot \Pr[\mathsf{Succ} \wedge \mathsf{Se} \wedge \mathsf{Bad}] - \Pr[\mathsf{Se}] - \tfrac{q_s}{2}\Pr[\mathsf{Ex} \vee \mathsf{Bad}] - (\tfrac{q_s}{2} - 1)\Pr[\mathsf{Se} \wedge \overline{\mathsf{Bad}}] + \tfrac{q_s}{2}\right| \\
= &\quad \mathsf{Adv}_P^{\mathsf{KE-fs}}(t', q_{\mathrm{ex}}) + \tfrac{q_s}{2} \cdot \mathsf{Adv}_P^{\mathsf{KE-fs}}(t', 1) + \left|2 \cdot \Pr[\mathsf{Succ} \wedge \mathsf{Se} \wedge \mathsf{Bad}] - \Pr[\mathsf{Se}] + \Pr[\mathsf{Se} \wedge \overline{\mathsf{Bad}}]\right| \\
= &\quad \mathsf{Adv}_P^{\mathsf{KE-fs}}(t', q_{\mathrm{ex}}) + \tfrac{q_s}{2} \cdot \mathsf{Adv}_P^{\mathsf{KE-fs}}(t', 1) + |2 \cdot \Pr[\mathsf{Succ} \wedge \mathsf{Se} \wedge \mathsf{Bad}] - \Pr[\mathsf{Se} \wedge \mathsf{Bad}]| \\
\leq &\quad \mathsf{Adv}_P^{\mathsf{KE-fs}}(t', q_{\mathrm{ex}}) + \tfrac{q_s}{2} \cdot \mathsf{Adv}_P^{\mathsf{KE-fs}}(t', 1) + \Pr[\mathsf{Bad}].
\end{aligned}
$$

Since $\Pr[\mathsf{Bad}] \leq \Pr[\mathsf{Forge}] + \Pr[\mathsf{Repeat}]$, this gives the desired result. ■

We remark that the above theorem is a generic result that applies to the invocation of the compiler on an *arbitrary* group KE protocol P. For specific protocols, a better exact security analysis may be obtainable.

4 A Constant-Round Group KE Protocol

In this section, we describe an efficient, two-round group KE protocol which achieves forward secrecy. Applying the compiler of the previous section to this protocol immediately yields (via Theorem 2) an efficient, three-round group AKE protocol achieving forward secrecy. The security of the present protocol is based on the decisional Diffie-Hellman (DDH) assumption [23], which we describe now. Let $\mathbb{G}$ be a cyclic group of prime order q and let g be a fixed generator of $\mathbb{G}$. Informally, the DDH problem is to distinguish between tuples of the form (g^x, g^y, g^{xy}) (called *Diffie-Hellman tuples*) and (g^x, g^y, g^z) where $z \neq xy$ is random (called *random tuples*); $\mathbb{G}$ is said to satisfy the DDH assumption if these two distributions are computationally indistinguishable. More formally, define $\mathsf{Adv}_{\mathbb{G}}^{\mathsf{ddh}}(t)$ as the maximum value, over all distinguishing algorithms D running in time at most t, of:

$$\left|\Pr[x, y \leftarrow \mathbb{Z}_q : D(g^x, g^y, g^{xy}) = 1] - \Pr[x, y \leftarrow \mathbb{Z}_q; z \leftarrow \mathbb{Z}_q \setminus \{xy\} : D(g^x, g^y, g^z) = 1]\right|,$$

where we assume that g is fixed and known to algorithm D. Given the above, $\mathbb{G}$ satisfies the DDH assumption if $\mathsf{Adv}_{\mathbb{G}}^{\mathsf{ddh}}(t)$ is "small" for "reasonable" values of t. (This definition is appropriate for a concrete security analysis; for asymptotic security one would consider an infinite sequence of groups $\mathcal{G} = \{\mathbb{G}_k\}_{k \geq 1}$ and require that $\mathsf{Adv}_{\mathbb{G}_k}^{\mathsf{ddh}}(t(k))$ be negligible in k for all polynomials t.) One standard way to generate a group assumed to satisfy the DDH assumption is to choose primes p, q such that $p = \beta q + 1$ and let $\mathbb{G}$ be the subgroup of order q in $\mathbb{Z}_p^*$. However, other choices of $\mathbb{G}$ are also possible.

The protocol presented here is essentially the protocol of Burmester and Desmedt [17], except that here $\mathbb{G}$ is a cyclic group of prime order assumed to satisfy the DDH assumption (in [17], $\mathbb{G}$ was taken to be an arbitrary cyclic group assumed to satisfy the *computational* Diffie-Hellman assumption). Our work was originally motivated by the fact that no proof of security appears in the proceedings version of [17]; furthermore, subsequent work in this area (e.g., [16, 13]) has implied that the Burmester-Desmedt protocol was "heuristic" and had not been proven secure. Subsequent to our work, however, we became aware that a proof of security for a variant of the Burmester-Desmedt protocol appears in the pre-proceedings of Eurocrypt '94 [22].[6] Even so, we note the following:

[6] We are happy to publicize this, especially since it appears to have been unknown to many others in the cryptographic community as well!

- Burmester and Desmedt show only that an adversary cannot compute the *entire* session key; in contrast, we show that the session key is indistinguishable from random.
- Burmester and Desmedt give a proof of security for their main protocol only for an *even* number of participants n. They introduce and prove secure a modified protocol (in which one player simulates the actions of two players) for the case of n odd.
- Finally, Burmester and Desmedt make no effort to optimize the concrete security of the reduction; indeed, this issue was not generally considered at that time.

For these reasons, we believe it is important to present a full proof of security for this protocol (taking care to achieve as tight a security reduction as possible) in a precise and widely-accepted model.

As required by our compiler, the protocol below ensures that players send every message to all members of the group via point-to-point links; although we refer to this as "broadcasting" we stress that no broadcast channel is assumed (in any case, the distinction is moot since we are dealing here with a passive adversary). For simplicity, in describing our protocol we assume a group $\mathbb{G}$ and a generator $g \in \mathbb{G}$ have been fixed in advance and are known to all parties in the network; however, this assumption can be avoided at the expense of an additional round in which the first player simply generates and broadcasts these values (that this remains secure follows from the fact that we are now considering a *passive* adversary). When n players $U_1, \ldots, U_n$ wish to generate a session key, they proceed as follows (the indices are taken modulo n so that player U_0 is U_n and player U_{n+1} is U_1):

Round 1 Each player U_i chooses a random $r_i \in \mathbb{Z}_q$ and broadcasts $z_i = g^{r_i}$.

Round 2 Each player U_i broadcasts $X_i = (z_{i+1}/z_{i-1})^{r_i}$.

Key computation Each player U_i computes their session key as:

$$K_i = (z_{i-1})^{nr_i} \cdot X_i^{n-1} \cdot X_{i+1}^{n-2} \cdots X_{i+n-2}.$$

It may be easily verified that all users compute the same key $g^{r_1r_2+r_2r_3+\cdots+r_nr_1}$.

Note that each user computes only three full-length exponentiations in $\mathbb{G}$ since $n \ll q$ in practice (typically, $q \approx 2^{160}$ while $n \ll 2^{32}$). We do not explicitly include sender identities and sequence numbers as required by the compiler of the previous section; however, as discussed there, it is easy to modify the protocol to include this information.

Theorem 3 *Protocol P is a secure group KE protocol achieving forward secrecy. Namely:*

$$\mathsf{Adv}_P^{\mathsf{KE-fs}}(t, q_{\mathrm{ex}}) \leq 4 \cdot \mathsf{Adv}_{\mathbb{G}}^{\mathsf{ddh}}(t') + \frac{6q_{\mathrm{ex}}}{|\mathbb{G}|},$$

where $t' = t + O(|\mathcal{P}| q_{\mathrm{ex}} t_{\mathrm{exp}})$ and t_{exp} is the time required to perform exponentiations in $\mathbb{G}$.

Proof Let $\varepsilon(t) \stackrel{\mathrm{def}}{=} \mathsf{Adv}_{\mathbb{G}}^{\mathsf{ddh}}(t)$. We consider first an adversary making a single $\mathsf{Execute}$ query and show that $\mathsf{Adv}_P^{\mathsf{KE-fs}}(t, 1) \leq 4 \cdot \varepsilon(t'')$, where $t'' = t + O(n \cdot t_{\mathrm{exp}})$ and n is the number of parties involved in the $\mathsf{Execute}$ query. We then discuss how to extend the proof for the case of multiple $\mathsf{Execute}$ queries without affecting the tightness of the security reduction.

Since there are no public keys in the protocol, we may ignore $\mathsf{Corrupt}$ queries. Assume an adversary $\mathcal{A}$ making a single query $\mathsf{Execute}(U_1, \ldots, U_n)$. We stress that the number of parties n is chosen by the adversary; since the protocol is symmetric and there are no public keys, however, the identities of the parties are unimportant for our discussion.

Let $n = 3s + k$ where $k \in \{3, 4, 5\}$ and $s \geq 0$ is an integer (we assume $n > 2$ since the protocol reduces to the standard Diffie-Hellman protocol [23] for the case $n = 2$). A detailed proof requires a slightly different analysis for each of the possible values of k and depending on whether or not $s = 0$; for ease of exposition, we describe here the proof for $k = 5, s > 0$ which is the most difficult case. In the real execution of the protocol, the distribution of the transcript T and the resulting session key sk is given by:

$$\mathsf{Real} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} r_1, \ldots, r_n \leftarrow \mathbb{Z}_q; \\ z_1 = g^{r_1},\ z_2 = g^{r_2},\ \ldots,\ z_n = g^{r_n}; \\ \Gamma_{1,2} = g^{r_1 r_2},\ \Gamma_{2,3} = g^{r_2 r_3},\ \ldots,\ \Gamma_{n-1,n} = g^{r_{n-1} r_n},\ \Gamma_{n,1} = g^{r_n r_1}; \\ X_1 = \frac{\Gamma_{1,2}}{\Gamma_{n,1}},\ X_2 = \frac{\Gamma_{2,3}}{\Gamma_{1,2}},\ \ldots,\ X_n = \frac{\Gamma_{n,1}}{\Gamma_{n-1,n}}; \\ \mathsf{T} = (z_1, \ldots, z_n, X_1, \ldots, X_n);\ \mathsf{sk} = (\Gamma_{n,1})^n \cdot (X_1)^{n-1} \cdots X_{n-1} \end{array} : (\mathsf{T}, \mathsf{sk}) \right\}.$$

Consider next the distribution Fake' defined as follows (recall $n = 3s + 5$ and $s \geq 1$):

$$\mathsf{Fake}' \stackrel{\text{def}}{=} \left\{ \begin{array}{l} r_1, \ldots, r_n \leftarrow \mathbb{Z}_q; \\ z_1 = g^{r_1},\ z_2 = g^{r_2},\ \ldots,\ z_n = g^{r_n}; \\ \Gamma_{1,2}, \Gamma_{2,3}, \Gamma_{3,4} \leftarrow \mathbb{G};\ \Gamma_{4,5} = g^{r_4 r_5} \\ \text{for } i = 1 \text{ to } s: \\ \quad \text{let } j = 3i + 3 \\ \quad \Gamma_{j-1,j} = g^{r_{j-1} r_j},\ \Gamma_{j,j+1} \leftarrow \mathbb{G},\ \Gamma_{j+1,j+2} = g^{r_{j+1} r_{j+2}}; \\ \Gamma_{n,1} = g^{r_n r_1}; \\ X_1 = \frac{\Gamma_{1,2}}{\Gamma_{n,1}},\ X_2 = \frac{\Gamma_{2,3}}{\Gamma_{1,2}},\ \ldots,\ X_n = \frac{\Gamma_{n,1}}{\Gamma_{n-1,n}}; \\ \mathsf{T} = (z_1, \ldots, z_n, X_1, \ldots, X_n);\ \mathsf{sk} = (\Gamma_{n,1})^n \cdot (X_1)^{n-1} \cdots X_{n-1} \end{array} : (\mathsf{T}, \mathsf{sk}) \right\}.$$

Claim *For any algorithm $\mathcal{A}$ running in time t we have:*

$$\left| \Pr[(\mathsf{T}, \mathsf{sk}) \leftarrow \mathsf{Real} : \mathcal{A}(\mathsf{T}, \mathsf{sk}) = 1] - \Pr[(\mathsf{T}, \mathsf{sk}) \leftarrow \mathsf{Fake}' : \mathcal{A}(\mathsf{T}, \mathsf{sk}) = 1] \right| \leq \varepsilon(t'') + \frac{1}{|\mathbb{G}|}.$$

Proof Given algorithm $\mathcal{A}$, consider the following algorithm D which takes as input a triple $(g_1, g_2, g_3) \in \mathbb{G}^3$ (where furthermore a generator $g \in \mathbb{G}$ is fixed): D generates $(\mathsf{T}, \mathsf{sk})$ according to distribution Dist', runs $\mathcal{A}(\mathsf{T}, \mathsf{sk})$, and outputs whatever $\mathcal{A}$ outputs. Distribution Dist' is defined as follows (note that this distribution depends on g_1, g_2, g_3):

$$\mathsf{Dist}' \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \alpha_0, \beta_0, \alpha'_0, \beta'_0, r_0, \alpha_1, \beta_1, \gamma_1, r_1, \ldots, \alpha_s, \beta_s, \gamma_s, r_s \leftarrow \mathbb{Z}_q; \\ z_1 = g^{\alpha_0} g_2^{\beta_0},\ z_2 = g_1,\ z_3 = g_2,\ z_4 = g^{\alpha'_0} g_1^{\beta'_0},\ z_5 = g^{r_0}; \\ \Gamma_{1,2} = g_1^{\alpha_0} g_3^{\beta_0},\ \Gamma_{2,3} = g_3,\ \Gamma_{3,4} = g_1^{\alpha'_0} g_3^{\beta'_0},\ \Gamma_{4,5} = z_4^{r_0}; \\ \text{for } i = 1 \text{ to } s: \\ \quad \text{let } j = 3i + 3 \\ \quad z_j = g^{\gamma_i} g_1,\ z_{j+1} = g^{\alpha_i} g_2^{\beta_i},\ z_{j+2} = g^{r_i}; \\ \quad \Gamma_{j-1,j} = z_j^{r_{i-1}},\ \Gamma_{j,j+1} = g_1^{\alpha_i} g_3^{\beta_i} z_{j+1}^{\gamma_i},\ \Gamma_{j+1,j+2} = z_{j+1}^{r_i}; \\ \Gamma_{n,1} = z_1^{r_s}; \\ X_1 = \frac{\Gamma_{1,2}}{\Gamma_{n,1}},\ X_2 = \frac{\Gamma_{2,3}}{\Gamma_{1,2}},\ \ldots,\ X_n = \frac{\Gamma_{n,1}}{\Gamma_{n-1,n}}; \\ \mathsf{T} = (z_1, \ldots, z_n, X_1, \ldots, X_n);\ \mathsf{sk} = (\Gamma_{n,1})^n \cdot (X_1)^{n-1} \cdots X_{n-1} \end{array} : (\mathsf{T}, \mathsf{sk}) \right\}.$$

We now analyze the output of D. On one hand, we have:

$$\left\{x, y \leftarrow \mathbb{Z}_q;\ g_1 = g^x, g_2 = g^y, g_3 = g^{xy};\ (\mathsf{T}, \mathsf{sk}) \leftarrow \mathsf{Dist}' : (\mathsf{T}, \mathsf{sk})\right\} \equiv \mathsf{Real},$$

representing the case when (g_1, g_2, g_3) is a Diffie-Hellman tuple. On the other hand, when D's input is a random tuple, the distributions Fake' and

$$\left\{x, y \leftarrow \mathbb{Z}_q;\ z \leftarrow \mathbb{Z}_q \setminus \{xy\};\ g_1 = g^x, g_2 = g^y, g_3 = g^z;\ (\mathsf{T}, \mathsf{sk}) \leftarrow \mathsf{Dist}' : (\mathsf{T}, \mathsf{sk})\right\}$$

are statistically close to within a factor of $\frac{1}{|\mathbb{G}|}$; the only difference is that in distribution Fake' the value of $\Gamma_{2,3}$ is chosen uniformly from $\mathbb{G}$ while in Dist' this value is chosen uniformly from $\mathbb{G} \setminus \{g_1^{\log_g g_2}\}$ (that the distributions are otherwise equivalent follows from random self-reducibility properties of the Diffie-Hellman problem; see [35] [5, Lemma 5.2]). Thus

$$\begin{aligned} &|\Pr[x, y \leftarrow \mathbb{Z}_q : D(g^x, g^y, g^{xy}) = 1] - \Pr[x, y \leftarrow \mathbb{Z}_q; z \leftarrow \mathbb{Z}_q \setminus \{xy\} : D(g^x, g^y, g^z) = 1]| \qquad (3) \\ &\quad \geq \left|\Pr[(\mathsf{T}, \mathsf{sk}) \leftarrow \mathsf{Real} : \mathcal{A}(\mathsf{T}, \mathsf{sk}) = 1] - \Pr[(\mathsf{T}, \mathsf{sk}) \leftarrow \mathsf{Fake}' : \mathcal{A}(\mathsf{T}, \mathsf{sk}) = 1]\right| - \frac{1}{|\mathbb{G}|}. \end{aligned}$$

Completing the proof, note that the running time of D is dominated by the running time of $\mathcal{A}$ and $O(n)$ exponentiations in $\mathbb{G}$; therefore, Expression (3) (which is D's advantage in solving the DDH problem) is at most $\varepsilon(t'')$. ☐

We now introduce one final distribution:

$$\mathsf{Fake} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} r_1, \ldots, r_n \leftarrow \mathbb{Z}_q; \\ z_1 = g^{r_1},\ \ldots,\ z_n = g^{r_n}; \\ \Gamma_{1,2}, \ldots, \Gamma_{n-1,n}, \Gamma_{n,1} \leftarrow \mathbb{G}; \\ X_1 = \frac{\Gamma_{1,2}}{\Gamma_{n,1}},\ X_2 = \frac{\Gamma_{2,3}}{\Gamma_{1,2}},\ \ldots,\ X_n = \frac{\Gamma_{n,1}}{\Gamma_{n-1,n}}; \\ \mathsf{T} = (z_1, \ldots, z_n, X_1, \ldots, X_n);\ \mathsf{sk} = (\Gamma_{n,1})^n \cdot (X_1)^{n-1} \cdots X_{n-1} \end{array} : (\mathsf{T}, \mathsf{sk}) \right\},$$

and prove the following claim:

Claim *For any algorithm $\mathcal{A}$ running in time t we have:*

$$\left|\Pr[(\mathsf{T}, \mathsf{sk}) \leftarrow \mathsf{Fake}' : \mathcal{A}(\mathsf{T}, \mathsf{sk}) = 1] - \Pr[(\mathsf{T}, \mathsf{sk}) \leftarrow \mathsf{Fake} : \mathcal{A}(\mathsf{T}, \mathsf{sk}) = 1]\right| \leq \varepsilon(t'').$$

Proof Given algorithm $\mathcal{A}$, consider the following algorithm D which takes as input a triple $(g_1, g_2, g_3) \in \mathbb{G}^3$ (where furthermore a generator $g \in \mathbb{G}$ is fixed): D generates $(\mathsf{T}, \mathsf{sk})$ according to distribution Dist, runs $\mathcal{A}(\mathsf{T}, \mathsf{sk})$, and outputs whatever $\mathcal{A}$ outputs. Distribution Dist is defined as follows (note that this distribution depends on g_1, g_2, g_3):

$$\mathsf{Dist} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} r_1, r_2, \alpha_0, \beta_0, \alpha'_0, \beta'_0, \gamma_0, \ldots, \alpha_s, \beta_s, \alpha'_s, \beta'_s, \gamma_s \leftarrow \mathbb{Z}_q; \\ z_1 = g^{\alpha_0} g_2^{\beta_0},\ z_2 = g^{r_1},\ z_3 = g^{r_2},\ z_4 = g^{\alpha'_0} g_2^{\beta'_0},\ z_5 = g^{\gamma_0} g_1; \\ \Gamma_{1,2}, \Gamma_{2,3}, \Gamma_{3,4} \leftarrow \mathbb{G};\ \Gamma_{4,5} = g_1^{\alpha'_0} g_3^{\beta'_0} z_4^{\gamma_0}; \\ \text{for } i = 1 \text{ to } s: \\ \quad \text{let } j = 3i + 3 \\ \quad z_j = g^{\alpha_i} g_2^{\beta_i},\ z_{j+1} = g^{\alpha'_i} g_2^{\beta'_i},\ z_{j+2} = g^{\gamma_i} g_1; \\ \quad \Gamma_{j-1,j} = g_1^{\alpha_i} g_3^{\beta_i} z_j^{\gamma_{i-1}},\ \Gamma_{j,j+1} \leftarrow \mathbb{G},\ \Gamma_{j+1,j+2} = g_1^{\alpha'_i} g_3^{\beta'_i} z_{j+1}^{\gamma_i}; \\ \Gamma_{n,1} = g_1^{\alpha_0} g_3^{\beta_0} z_1^{\gamma_s}; \\ X_1 = \frac{\Gamma_{1,2}}{\Gamma_{n,1}},\ X_2 = \frac{\Gamma_{2,3}}{\Gamma_{1,2}},\ \ldots,\ X_n = \frac{\Gamma_{n,1}}{\Gamma_{n-1,n}}; \\ \mathsf{T} = (z_1, \ldots, z_n, X_1, \ldots, X_n);\ \mathsf{sk} = (\Gamma_{n,1})^n \cdot (X_1)^{n-1} \cdots X_{n-1} \end{array} : (\mathsf{T}, \mathsf{sk}) \right\}.$$

Arguing as in the previous claim (and again using the self-reducibility of the DDH problem), we have:

$$\begin{aligned}
&\left|\Pr[(\mathsf{T},\mathsf{sk}) \leftarrow \mathsf{Fake}' : \mathcal{A}(\mathsf{T},\mathsf{sk}) = 1] - \Pr[(\mathsf{T},\mathsf{sk}) \leftarrow \mathsf{Fake} : \mathcal{A}(\mathsf{T},\mathsf{sk}) = 1]\right| \\
&\quad = \ \left|\Pr[x, y \leftarrow \mathbb{Z}_q : D(g^x, g^y, g^{xy}) = 1] - \Pr[x, y \leftarrow \mathbb{Z}_q; z \leftarrow \mathbb{Z}_q \setminus \{xy\} : D(g^x, g^y, g^z) = 1]\right| \\
&\quad \leq \ \varepsilon(t'').
\end{aligned}$$

□

In experiment Fake, let $w_{i,i+1} \stackrel{\text{def}}{=} \log_g \Gamma_{i,i+1}$ for $1 \leq i \leq n$. Given T, the values $w_{1,2}, \ldots, w_{n,1}$ are constrained by the following n equations (only $n-1$ of which are linearly independent):

$$\begin{aligned}
\log_g X_1 &= w_{1,2} - w_{n,1} \\
&\vdots \\
\log_g X_n &= w_{n,1} - w_{n-1,n}.
\end{aligned}$$

Furthermore, $\mathsf{sk} = g^{w_{1,2}+w_{2,3}+\cdots+w_{n,1}}$; equivalently, we have

$$\log_g \mathsf{sk} = w_{1,2} + w_{2,3} + \cdots + w_{n,1}.$$

Since this final equation is linearly independent from the set of equations above, sk is independent of T. This implies that, for any computationally-unbounded adversary $\mathcal{A}$:

$$\Pr[(\mathsf{T}, \mathsf{sk}_0) \leftarrow \mathsf{Fake}; \mathsf{sk}_1 \leftarrow \mathbb{G}; b \leftarrow \{0,1\} : \mathcal{A}(\mathsf{T}, \mathsf{sk}_b) = b] = 1/2.$$

Combining this with the previous two claims shows that $\mathsf{Adv}_P^{\mathsf{KE-fs}}(t, 1) \leq 4 \cdot \varepsilon(t'') + 2/|\mathbb{G}|$.

For the case of $q_{\rm ex} > 1$, a standard hybrid argument immediately shows that

$$\mathsf{Adv}_P^{\mathsf{KE-fs}}(t, q_{\rm ex}) \leq q_{\rm ex} \cdot \mathsf{Adv}_P^{\mathsf{KE-fs}}(t, 1).$$

Yet tighter concrete security can be obtained by again using the random self-reducibility of the DDH problem [5, Lemma 5.2]. In particular, given a tuple $(g_1, g_2, g_3) \in \mathbb{G}^3$ and a fixed generator g, one can efficiently generate $q_{\rm ex}$ tuples $L = \{(g_1^1, g_2^1, g_3^1), \ldots, (g_1^{q_{\rm ex}}, g_2^{q_{\rm ex}}, g_3^{q_{\rm ex}})\}$ such that (1) if (g_1, g_2, g_3) is a Diffie-Hellman tuple, then all tuples in L are Diffie-Hellman tuples whose first two components are randomly distributed in $\mathbb{G}^2$ (independently of anything else); (2) if (g_1, g_2, g_3) is a random tuple, then all tuples in L are randomly distributed in $\mathbb{G}^3$ (again, independently of anything else). In the second case, with all but probability $\frac{q_{\rm ex}}{|\mathbb{G}|}$ it will be the case that $\log_g g_3^i \neq \log_g g_1^i \cdot \log_g g_2^i$ for all i.

Paralleling the preceding proof, we may define distributions $\mathsf{Real}_{q_{\rm ex}}$, $\mathsf{Fake}'_{q_{\rm ex}}$, $\mathsf{Dist}'_{q_{\rm ex}}$, $\mathsf{Fake}_{q_{\rm ex}}$, and $\mathsf{Dist}_{q_{\rm ex}}$ which simply consist of $q_{\rm ex}$ (independent) copies of each of the corresponding distributions; in the case of $\mathsf{Dist}'_{q_{\rm ex}}$ and $\mathsf{Dist}_{q_{\rm ex}}$ we use the corresponding tuple (g_1^i, g_2^i, g_3^i) for the $i^{\rm th}$ copy. Corresponding to the first claim, one could then show that for any algorithm $\mathcal{A}$ running in time t:

$$\left|\Pr[(\vec{\mathsf{T}}, \vec{\mathsf{sk}}) \leftarrow \mathsf{Real}_{q_{\rm ex}} : \mathcal{A}(\vec{\mathsf{T}}, \vec{\mathsf{sk}}) = 1] - \Pr[(\vec{\mathsf{T}}, \vec{\mathsf{sk}}) \leftarrow \mathsf{Fake}'_{q_{\rm ex}} : \mathcal{A}(\vec{\mathsf{T}}, \vec{\mathsf{sk}}) = 1]\right| \leq \varepsilon(t') + \frac{2q_{\rm ex}}{|\mathbb{G}|},$$

where t' is as in the statement of the theorem. Corresponding to the second claim, one could show that for ant algorithm $\mathcal{A}$ running in time t:

$$\left|\Pr[(\vec{\mathsf{T}}, \vec{\mathsf{sk}}) \leftarrow \mathsf{Fake}'_{q_{\rm ex}} : \mathcal{A}(\vec{\mathsf{T}}, \vec{\mathsf{sk}}) = 1] - \Pr[(\vec{\mathsf{T}}, \vec{\mathsf{sk}}) \leftarrow \mathsf{Fake}_{q_{\rm ex}} : \mathcal{A}(\vec{\mathsf{T}}, \vec{\mathsf{sk}}) = 1]\right| \leq \varepsilon(t') + \frac{q_{\rm ex}}{|\mathbb{G}|}.$$

Finally, using the same techniques as above, it is straightforward to see that even for a computationally-unbounded $\mathcal{A}$

$$\Pr[(\vec{\mathsf{T}}, \vec{\mathsf{sk}}_0) \leftarrow \mathsf{Fake}; \vec{\mathsf{sk}}_1 \leftarrow \mathbb{G}^{q_{\rm ex}}; b \leftarrow \{0,1\} : \mathcal{A}(\mathsf{T}, \vec{\mathsf{sk}}_b) = b] = 1/2.$$

Putting these together gives the result stated by the theorem. ■

References

[1] Y. Amir, Y. Kim, C. Nita-Rotaru, and G. Tsudik. On the Performance of Group Key Agreement Protocols. *Proc. 22nd International Conference on Distributed Computing Systems*, IEEE, 2002, pp. 463–464. Full version available at http://www.cnds.jhu.edu/publications/.

[2] G. Ateniese, M. Steiner, and G. Tsudik. Authenticated Group Key Agreement and Friends. *Proc. 5th Annual ACM Conference on Computer and Communications Security*, ACM, 1998, pp. 17–26.

[3] G. Ateniese, M. Steiner, and G. Tsudik. New Multi-Party Authentication Services and Key Agreement Protocols. *IEEE Journal on Selected Areas in Communications* 18(4): 628–639 (2000).

[4] C. Becker and U. Wille. Communication Complexity of Group Key Distribution. *Proc. 5th Annual ACM Conference on Computer and Communication Security*, ACM, 1998, pp. 1–6.

[5] M. Bellare, A. Boldyreva, and S. Micali. Public-Key Encryption in a Multi-user Setting: Security Proofs and Improvements. *Advances in Cryptology — Eurocrypt 2000*, LNCS vol. 1807, B. Preneel ed., Springer-Verlag, 2000, pp. 259–274.

[6] M. Bellare, R. Canetti, and H. Krawczyk. A Modular Approach to the Design and Analysis of Authentication and Key Exchange Protocols. *Proc. 30th Annual ACM Symposium on Theory of Computing*, ACM, 1998, pp. 419–428.

[7] M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated Key Exchange Secure Against Dictionary Attacks. *Advances in Cryptology — Eurocrypt 2000*, LNCS vol. 1807, B. Preneel ed., Springer-Verlag, 2000, pp. 139–155.

[8] M. Bellare and P. Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. *Proc. 1st Annual ACM Conference on Computer and Communications Security*, ACM, 1993, pp. 62–73.

[9] M. Bellare and P. Rogaway. Entity Authentication and Key Distribution. *Advances in Cryptology — Crypto '93*, LNCS vol. 773, D.R. Stinson ed., Springer-Verlag, 1993, pp. 232–249.

[10] M. Bellare and P. Rogaway. Provably-Secure Session Key Distribution: the Three Party Case. *Proc. 27th Annual ACM Symp. on Theory of Computing*, ACM, 1995, pp. 57–66.

[11] R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kutten, R. Molva, and M. Yung. Systematic Design of Two-Party Authentication Protocols. *IEEE Journal on Selected Areas in Communications* 11(5): 679–693 (1993).

[12] C. Boyd. On Key Agreement and Conference Key Agreement. *Australasian Conference on Information Security and Privacy — ACISP '97*, LNCS vol. 1270, V. Varadharajan, J. Pieprzyk, and Y. Mu eds., Springer-Verlag, 1997, pp. 294–302.

[13] C. Boyd and J.M.G. Nieto. Round-Optimal Contributory Conference Key Agreement. *Public-Key Cryptography*, LNCS vol. 2567, Y. Desmedt ed., Springer-Verlag, 2003, pp. 161–174.

[14] E. Bresson, O. Chevassut, and D. Pointcheval. Provably Authenticated Group Diffie-Hellman Key Exchange — The Dynamic Case. *Advances in Cryptology — Asiacrypt 2001*, LNCS vol. 2248, C. Boyd ed., Springer-Verlag, 2001, pp. 290–309.

[15] E. Bresson, O. Chevassut, and D. Pointcheval. Dynamic Group Diffie-Hellman Key Exchange under Standard Assumptions. *Advances in Cryptology — Eurocrypt 2002*, LNCS vol. 2332, L. Knudsen ed., Springer-Verlag, 2002, pp. 321–336.

[16] E. Bresson, O. Chevassut, D. Pointcheval, and J.-J. Quisquater. Provably Authenticated Group Diffie-Hellman Key Exchange. *Proc. 8th Annual ACM Conference on Computer and Communications Security*, ACM, 2001, pp. 255–264.

[17] M. Burmester and Y. Desmedt. A Secure and Efficient Conference Key Distribution System. *Advances in Cryptology — Eurocrypt '94*, LNCS vol. 950, A. De Santis, ed., Springer-Verlag, 1995, pp. 275–286.

[18] R. Canetti, O. Goldreich, and S. Halevi. The Random Oracle Methodology, Revisited. *Proc. 30th Annual ACM Symp. on Theory of Computing*, ACM, 1998, pp. 209–218.

[19] R. Canetti and H. Krawczyk. Key-Exchange Protocols and Their Use for Building Secure Channels. *Advances in Cryptology — Eurocrypt 2001*, LNCS vol. 2045, B. Pfitzmann, ed., Springer-Verlag, 2001, pp. 453–474.

[20] R. Canetti and H. Krawczyk. Universally Composable Notions of Key Exchange and Secure Channels. *Advances in Cryptology — Eurocrypt 2002*, LNCS vol. 2332, L. Knudsen, ed., Springer-Verlag, 2002, pp. 337–351.

[21] R. Canetti and H. Krawczyk. Security Analysis of IKE's Signature-Based Key-Exchange Protocol. *Advances in Cryptology — Crypto 2002*, LNCS vol. 2442, M. Yung, ed., Springer-Verlag, 2002, pp. 143–161.

[22] Y. Desmedt. Personal communication (including a copy of the pre-proceedings version of [17]), March 2003.

[23] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory* 22(6): 644–654 (1976).

[24] W. Diffie, P. van Oorschot, and M. Wiener. Authentication and Authenticated Key Exchanges. *Designs, Codes, and Cryptography* 2(2): 107–125 (1992).

[25] M. Fischer, N. Lynch, and M. Patterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM* 32(2): 374–382 (1985).

[26] I. Ingemarsson, D.T. Tang, and C.K. Wong. A Conference Key Distribution System. *IEEE Transactions on Information Theory* 28(5): 714–720 (1982).

[27] M. Just and S. Vaudenay. Authenticated Multi-Party Key Agreement. *Advances in Cryptology — Asiacrypt 1996*, LNCS vol. 1163, K. Kim and T. Matsumoto, eds., Springer-Verlag, 1996, pp. 36–49.

[28] J. Katz, R. Ostrovsky, and M. Yung. Forward Secrecy in Password-Only Key-Exchange Protocols. *Security in Communication Networks (SCN 2002)*, LNCS vol. 2576, S. Cimato, C. Galdi, and G. Persiano, eds., Springer-Verlag, 2002, pp. 29–44.

[29] Y. Kim, A. Perrig, and G. Tsudik. Simple and Fault-Tolerant Key Agreement for Dynamic Collaborative Groups. *Proc. 7th Annual ACM Conference on Computer and Communication Security*, ACM, 2000, pp. 235–244.

[30] Y. Kim, A. Perrig, and G. Tsudik. Communication-Efficient Group Key Agreement. *Proc. IFIP TC11 16th Annual Working Conference on Information Security (IFIP/SEC)*, Kluwer, 2001, pp. 229–244.

[31] H. Krawczyk. SKEME: A Versatile Secure Key-Exchange Mechanism for the Internet. *Proc. of the Internet Society Symposium on Network and Distributed System Security*, Feb. 1996, pp. 114–127.

[32] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[33] A. Mayer and M. Yung. Secure Protocol Transformation via "Expansion": From Two-Party to Groups. *Proc. 6th ACM Conference on Computer and Communication Security*, ACM, 1999, pp. 83–92.

[34] O. Pereira and J.-J. Quisquater. A Security Analysis of the Cliques Protocol Suites. *Proc. 14th IEEE Computer Security Foundations Workshop*, IEEE, 2001, pp. 73–81.

[35] V. Shoup. On Formal Models for Secure Key Exchange. Draft, 1999. Available at `http://eprint.iacr.org/1999/012`.

[36] D. Steer, L. Strawczynski, W. Diffie, and M. Wiener. A Secure Audio Teleconference System. *Advances in Cryptology — Crypto '98*, LNCS vol. 403, Springer-Verlag, 1990, pp. 520–528.

[37] M. Steiner, G. Tsudik, and M. Waidner. Key Agreement in Dynamic Peer Groups. *IEEE Trans. on Parallel and Distributed Systems* 11(8): 769–780 (2000).

[38] W.-G. Tzeng. A Practical and Secure Fault-Tolerant Conference Key Agreement Protocol. *Public-Key Cryptography*, LNCS vol. 1751, H. Imai and Y. Zheng, eds., Springer-Verlag, 2000, pp. 1-13.

[39] W.-G. Tzeng and Z.-J. Tzeng. Round Efficient Conference Key Agreement Protocols with Provable Security. *Advances in Cryptology — Asiacrypt 2000*, LNCS vol. 1976, T. Okamoto, ed., Springer-Verlag, 2000, pp. 614–628.