On Ends-to-Ends Encryption:
Asynchronous Group Messaging with Strong Security Guarantees

Katriel Cohn-Gordon[1], Cas Cremers[1], Luke Garratt[1], Jon Millican[2], and Kevin Milner[1]

[1]Department of Computer Science, University of Oxford
[2]Facebook

July 2017

Abstract

In the past few years secure messaging has become mainstream, with over a billion active users of end-to-end encryption protocols through apps such as WhatsApp, Signal, Facebook Messenger, Google Allo, Wire and many more. While these users' two-party communications now enjoy very strong security guarantees, it turns out that many of these apps provide, without notifying the users, a weaker property for *group* messaging: an adversary who compromises a single group member can intercept communications indefinitely.

One reason for this discrepancy in security guarantees is that most existing group messaging protocols are fundamentally *synchronous*, and thus cannot be used in the asynchronous world of mobile communications. In this paper we show that this is not necessary, presenting a design for a tree-based group key exchange protocol in which no two parties ever need to be online at the same time. Our design achieves strong security guarantees, in particular including post-compromise security.

We give a computational security proof for our core design as well as a proof-of-concept implementation, showing that it scales efficiently even to large groups. Our results show that strong security guarantees for group messaging are achievable even in the modern, asynchronous setting, without resorting to using inefficient point-to-point communications for large groups. By building on standard and well-studied constructions, our hope is that many existing solutions can be applied while still respecting the practical constraints of mobile devices.

1 Introduction

The level of security offered by secure messaging systems has improved substantially over recent years; for example, WhatsApp now provides end-to-end encryption for its billion active users, based on Open Whisper Systems' Signal Protocol (WhatsApp, 2016; Marlinspike, 2016), and the Guardian publishes Signal contact details for its investigative journalism teams (Guardian, 2017). An important constraint of modern messaging systems, compared to related protocols such as those used for key exchange, is that they must allow for *asynchronous communication*: Alice must be able to send a message to Bob even if Bob is currently offline. Typically, the encrypted message is temporarily stored on a (possibly untrusted) server, to be delivered to Bob once he comes online again.

This asynchronicity constraint implies that standard solutions to achieve, e.g., perfect forward secrecy, such as a Diffie–Hellman (DH) key exchange, do not apply directly. This has driven the development of novel techniques to achieve perfect forward secrecy without interaction, for example using sets of "prekeys" (Marlinspike, 2013) that Bob uploads to a server, essentially serving as precomputed DH keys, or by using puncturable encryption (Green and Miers, 2015).

Moreover, some modern messaging protocols offer a property called post-compromise security (PCS) (Cohn-Gordon et al., 2016), often referred to as "future secrecy" or "self-healing": even after Alice's device is entirely compromised by an adversary, revealing her long-term key and potentially all random values generated so far, she may be able to later regain secure communications with others, as long as she has one exchange with them in which the adversary does not interfere. PCS limits the scope of a compromise, forcing an adversary to act as a permanent man-in-the-middle if they wish to exploit knowledge of a long-term key. Thus far, PCS-style properties have only been proven for point-to-point protocols.

In practice however, point-to-point communication does not suffice for real-world messaging applications, in which group and multi-device messaging are often important features. In theory, it is easy to solve this: Alice

K.C-G. thanks Merton College and the Oxford CDT in Cyber Security for their support.

uses the point-to-point protocol with each of her communication partners. However, as group sizes become larger, this leads to inefficient systems in which the bandwidth and computational cost for sending a message grows linearly with the group size (as each recipient gets their own, differently encrypted, copy of the message). In many real-world scenarios, this inefficiency can be problematic, especially in areas with restricted bandwidth or high data costs (e.g., 2G networks in the developing world). The 2015 State of Connectivity report by `internet.org` (internet.org, 2016) lists affordability of mobile data as one of the four major barriers to global connectivity, with a developing-world average monthly data use of just 255 MB/device.

Instead of using a point-to-point protocol with each group member, a theoretical alternative is to use a group protocol Kim et al. (2004); Brecher et al. (2009); Desmedt et al. (2007); Lee et al. (2003); Kim et al. (2000, 2001); Bresson et al. (2001). These typically use tree structures based on DH keys to combine the participants' individual keys into a group key. This reduces both the computational effort and bandwidth required to send a message, as the sender sends only one copy of each message encrypted under the group key. However, such protocols are in general not asynchronous, and do not consider post-compromise security—they do not make any guarantees after the adversary completely compromises a participant.

The lack of asynchronicity, among other considerations, means that modern messaging protocols which provide post-compromise security for two-party communications generally drop this guarantee for their group messaging implementations without notifying the users. For example, WhatsApp, Facebook Messenger and the Signal app have mechanisms that aim to achieve post-compromise security for two-party communications, but for conversations containing three or more devices they use a simpler key-transport mechanism ("sender keys") which does not achieve it (Facebook, 2017; WhatsApp, 2016). Indeed, in all three, an adversary who fully compromises a single group member can indefinitely and passively read future communications in that group (though certain events, such as new device registration, may cause the group to change and new keys to be generated). In practice this means that in these apps, if a third party is added to a two-party communication, the security of the communication is decreased without informing the users.

The question thus arises: is there a secure, end-to-end encrypted group messaging solution that

(i) allows participants to communicate *asynchronously*,
(ii) *does not require* point-to-point communications, and
(iii) admits *strong security guarantees* such as post-compromise security?

In this paper we address this open question, and show how to devise a protocol that achieves it.

Contributions

Our main contributions are the following:

We design a fully-asynchronous tree-based group key exchange protocol that offers modern strong security properties. The protocol derives a group key for a set of agents without any pair needing to be online at the same time. Modern messaging protocols must work fully asynchronously, and our design enables this.

We give a game-based computational security model for our protocol, building on multi-stage models to capture the key updating property. This allows us to encode strong security properties such as post-compromise security: even after total compromise, it is possible for an agent to participate in a secure group key exchange.

We give a game-hopping computational proof of the unauthenticated core of our protocol, with an explicit reduction to the decisional DH problem, and a symbolic verification of its authentication property. Our hybrid argument follows the style of e.g. (Kobeissi et al., 2017).

We present a proof-of-concept Java implementation of all of our core algorithms, increasing confidence in the functional correctness and feasibility of our design.

Our design approach is of independent interest beyond our specific construction. In particular, by using simple and well-studied constructions, our design allows many insights from the existing literature in (synchronous) group protocols to be applied in the asynchronous setting.

2 Background

2.1 Other Group Messaging Protocols

2.1.1 OTR-style

Goldberg et al. (2009) define Multi-Party Off the Record Messaging (mpOTR) as a generalisation of the classic OTR (Borisov et al., 2004) protocol, aiming for security and deniability in online messaging. mpOTR has since given rise to a number of interactive protocols, such as `eQualit.ie`'s $(N+1)SEC$ (eQualit.ie, 2016).

Table 1: Asymptotic efficiencies and properties of some group messaging solutions as a function of the group size n, both in the setup phase and for each message sent. Sender Keys is the design currently used to power Whatsapp, Signal and Facebook Secret Conversation group messaging.

		exponentiations	encryptions/ server storage/ bandwidth	local storage	provable post-compromise security
sender keys (Signal)	setup	$4n-4$	$n-1$	$n \times$ channels + $n \times$ keys	✗
	ongoing	0	1		
Signal	setup	$4n-4$	1	$n \times$ channel	✓
	ongoing	n to $2n$	$n-1$		
our solution	setup	$\log n$ ($2n$ for initiator)	$2n$	$2\log n$ keys	✓
	ongoing	$2\log n$	1		

The general design of this family of protocols is as follows. First, parties conduct a number of interactive rounds of communication in order to derive a group key. Second, parties communicate online, perhaps performing additional cryptographic operations. Finally, there may be a closing phase (for instance, to assess transcript consistency between all participants).

All of these protocols are intrinsically synchronous: they require all parties to come online at the same time for the initial key exchange. This is not a problem in their context of XMPP-style instant messaging, but does not work for mobile and unreliable networks.

2.1.2 Sender Keys

If participants have secure pairwise channels, they can send encrypted "broadcast" keys to each group member separately, and then broadcast their messages encrypted under those keys. This is implemented in `libsignal` as the "Sender Keys" variant of the Signal Protocol. However, it sacrifices some of the strong security properties achieved by the Double Ratchet: if an adversary ever learns a sender key, it can subsequently eavesdrop on all messages and impersonate the key's owner in the group, even though it cannot do so over the pairwise Signal channels (whose keys are continuously updated).

Regularly broadcasting new sender keys over the secure pairwise channels prevents this type of attack. However, since a new sender key message must be sent separately to each group member, this scales linearly in the size of the group for a given key rotation frequency.

2.1.3 n-party DH

Perhaps the most natural generalisation of DH key updates to n parties would be a primitive that allows for the following: given all of $\mathsf{pk}_1, \ldots, \mathsf{pk}_n$ and a single sk_i ($i \in [n]$), derive a value grk which is hard to compute without knowing one of the sk_i. With $n = 2$ this can be achieved by traditional DH, and with $n = 3$ Joux (2004) gives a pairing-based construction.

However, for general n construction of such a primitive is a known open problem. Boneh and Silverberg (2003) essentially generalise the Joux protocol with a construction from an $(n-1)$-non-degenerate linear map on the integers. Boneh and Zhandry (2014) present one from indistinguishability obfuscation (iO), and recent work by Ma and Zhandry (2017) formalises the concept as an "encryptor combiner" and gives constructions from iO or from certain lattice assumptions.

2.1.4 Tree-based group DH

There is a very large body of literature on tree-based group key agreement schemes. An early example is the "audio teleconference system" of Steer et al. (1990), and the seminal academic work is perhaps Wallner et al. (1999) or Wong et al. (2000). Later examples include (Kim et al., 2004; Brecher et al., 2009; Desmedt et al., 2007; Lee et al., 2003; Kim et al., 2000, 2001; Yang et al., 2017; Chen and Tzeng, 2017), among many others. Roughly, these protocols assign private DH keys to leaves of a binary tree, defining

(i) g^{xy} as the secret key of a node whose two children have secret keys x and y, and

(ii) g^z as the public ('blinded') key of a node with secret key z.

Recursively computing secret keys through the tree, starting from the leaves, yields a value at the root which we call the "tree key", with the property that it can only be computed with knowledge of at least one secret leaf key. A similar construction can be done with ternary trees for the three-party Joux protocol.

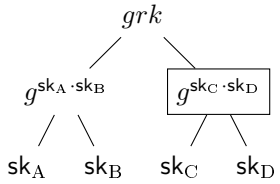
In order to compute the secret key $g^{xy} = (g^y)^x$ assigned to a non-leaf node, an agent must know the secret key x of one of its children and the public key g^y of the other. Thus, to compute the tree key requires an agent to know

(i) one secret leaf key λ_j, and

(ii) all public node keys pk_1 to pk_n along its *copath*,

where the copath of a node is the list of sibling nodes along its path to the tree root. The group key is computed by alternately exponentiating the next public key with the current secret, and applying an injection from group elements to integers.

Interactivity in these protocols comes from, at least in part, the requirement for Alice to know the public keys on her copath. For example, in the figure below we depict a small DH tree of size four. Even if Alice knows the public keys corresponding to each leaf node, to derive the tree key she needs to know the public key $g^{\mathsf{sk_C} \cdot \mathsf{sk_D}}$ of the boxed node, which she cannot compute herself.



Traditionally, one of C or D is chosen by the messaging system to compute $g^{\mathsf{sk_C} \cdot \mathsf{sk_D}}$ and broadcast its public key. For example, Kim et al. (2004) describe a system where certain agents act as "sponsors" for their subtrees and broadcast the public keys which they know.

2.2 Widely-Used Implementations

Various widely-used mobile apps have deployed encrypted group messaging protocols, and we survey some of the most popular.

2.2.1 WhatsApp

WhatsApp implements end-to-end encryption for group messaging using the Sender Keys variant of Signal for all groups of size 3+, using the existing support for Signal in pairwise channels. Sender keys are rotated whenever a participant is removed from a group but otherwise are never changed; as discussed above, an adversary who learns a sender key can therefore impersonate or eavesdrop on its owner until the group changes.

WhatsApp also supports multiple devices for a single user. To do so, it defines the mobile phone as a master device and allows secondary devices to connect by scanning a QR code. When Alice sends a message from a secondary device, WhatsApp first sends the message to her mobile phone, and then over the pairwise Signal channel to the intended peer. While this method does allow for multiple device functionality, it suffers from the downside that if Alice's phone is offline then she cannot use WhatsApp, even if her other device is connected to the Internet.

2.2.2 Facebook Messenger Secret Conversations

Secret Conversations on Facebook Messenger similarly use the Sender Keys variant of Signal for all conversations involving 3+ devices. As in the WhatsApp implementation, Sender Keys are only rotated when a device is removed from a conversation, so compromising a sender key will allow future messages from that user to be eavesdropped.

2.2.3 Signal

The Signal mobile application also uses the Sender Keys variant for group messaging. Signal allows multi-device messaging by allowing a mobile phone to provision the desktop app, similarly to WhatsApp. However additional devices on a Signal account are first class participants in that they use Signal Protocol directly, as opposed to routing messages via the phone.

2.2.4 iMessage

Apple's iMessage implements group messaging using pairwise channels: one copy of each message is encrypted and sent for each group member over pairwise encrypted channels. We remark that this indicates that in a group of size n, performing $\sim 2n$ asymmetric operations per message was considered a practical cost on an iPhone 3GS (circa 2009).

3 Objectives

Security properties for authenticated key exchange (AKE) protocols are extremely well-studied. In this section, we describe a high-level threat model and security goals for group messaging protocols.

3.1 Security Goals

3.1.1 Secrecy and Authentication

Our fundamental goal is confidentiality and authenticity of keys: if Alice exchanges a key with Bob, even an active network adversary should not be able to learn that key.

3.1.2 Post-Compromise Security

Traditional security models do not provide any guarantees *after* the long-term keys of a participant are compromised: it is not considered an attack to learn Bob's identity key and then impersonate him to Alice. Cohn-Gordon et al. (2016) defined the notion of *post-compromise security* to cover this scenario, showing that it is achievable through the use of persistent protocol state.

We aim explicitly to achieve a form of PCS in our messaging protocols. Specifically, if the full state of a group member is compromised (including their long-term keys as well as any others which they may have derived) but the group conversation then continues without interference and new, uncompromised keys are derived, then the resulting group key should once again be secret.

Absent this goal, many simpler designs are possible. In particular, the "sender keys" variant of Signal meets our other criteria; its weakness is that learning a sender key enables the computation of all future message keys, and hence it does not meet this form of PCS. The PCS property is a major distinguishing feature of modern two-party messaging protocols, and offers significant protection to adversaries with large resources, as it forces them to actively interfere in all sessions even after they manage to temporarily compromise a device. We therefore believe that this property is important for modern protocols.

3.1.3 Imperfect randomness

Security models such as extended Canetti-Krawczyk and its generalisations (LaMacchia et al., 2007; Cremers and Feltz, 2012) allow for the corruption of random numbers generated by a party as long as their long-term keys remain secure. However, many widely-used protocols do not to achieve this property—for example, neither TLS nor Signal are secure if both sides' ephemeral keys are revealed. We aim for security even if some random numbers are revealed, as long as not all (in the current tree) are.

3.2 Other Types of Property

3.2.1 Sender-specific authentication

In a group, authentication becomes more subtle: if Alice, Bob and Charlie share a symmetric key and Alice receives a message encrypted under it which she did not sent, she can conclude only that either Bob or Charlie sent it. Depending on the context, this may not be a desirable property of a group messaging system—in OTR it is considered a feature as a form of deniability, while in Signal Protocol it is ruled out by distributing individual signature keys used to authenticate messages which are encrypted with the group key. We choose the simpler design and do not include signature keys, discussing sender-specific authentication further in Section 8.

Centralised, unencrypted group messaging systems usually provide individual authentication via the service provider's accounts—for example, Facebook Messenger group chats do not allow Bob to impersonate Charlie, because Bob must log into his Facebook account to send a message as himself. We do not assume such a trusted third party in our analyses. Of course, an encrypted messaging system can *also* include authentication from a third party, as with e.g. Facebook Messenger's Secret Conversations.

3.2.2 Malicious group members

In the two-party case, traditional security properties are generally of the form "if the peer to a session is honest then P". With n parties, there is an intermediate type of property: "if $m < n$ members of the group are honest then P". For example, Abdalla et al. (2010) give a group key exchange protocol which enables subsets of the group to derive their own key, aiming for security in a subset even if another group member is malicious.

Although these properties are useful, we consider them orthogonal to our core research question. Moreover, because we use standard constructions from the (synchronous) literature, we anticipate that extending our designs to handle group membership changes should be relatively straightforward. We discuss dynamic groups further in Section 8.

3.2.3 Executability

Implementations of group messaging systems must deal with desynchronisation of state: if Bob attempts to update his state without realising that Alice has already performed an update which he does not know about, he may lose track of the current group key. This does not violate any secrecy properties, but breaks availability.

In general, the standard solution is at the transport layer, either by enforcing in-order message delivery or by refusing to accept out-of-order key updates, instead delivering the latest group state. This solution works fine for many group sizes, but in very large groups may cause a performance bottleneck at the transport layer. (Chen and Tzeng, 2017) study this problem in more detail; we consider it out of scope for our work.

3.2.4 Transcript agreement

In many scenarios it is valuable for all group participants agree on the ordered list of messages that were sent and received in the group. Although this is a useful property, it has many subtleties that are orthogonal to our key research questions and we do not deal with it in this paper.

3.3 Security properties

Informally, we want our messaging protocol to provide *implicit authentication* and *message secrecy* under a variety of strong adversary scenarios:

Security under a network (Dolev-Yao) adversary. The adversary has full control of message delivery, able to intercept, read and modify any messages sent over the network.

Forward secrecy. Once a stage has derived a key, revealing long-term keys or any random values from subsequent stages should not compromise its security.

Post-compromise secrecy (Cohn-Gordon et al., 2016). If a stage derives a key, but at least one previous stage was uncompromised, the derived key should be secret. Equivalently, after all of a party's secrets are compromised, if an intermediate stage completes with an uncorrupted key, then all subsequent stages should be secure.

4 Notation

Trees

We defined binary trees as a combination of nodes with two nested children and leaves with no children, along with associated data at each element:

$$\text{tree} \rightarrow (\text{node}(\text{tree}, \text{tree}), \cdot) \,|\, (\text{leaf}, \cdot).$$

For a binary tree T, we use the notation $|T|$ to refer to the total number of leaves in the tree. We label each element of a tree with an index pair (x, y), where x represents the level of the element: the number of nodes (including the element itself) in the path to the root at index $(0, 0)$. The children of a node at index (x, y) are $(x + 1, 2y)$ and $(x + 1, 2y + 1)$; thus y represents the index a node would have in its level if the tree were complete. To refer to the associated data at a tree index (x, y) in a tree T, we write $T_{x,y}$.

All tree elements but the root have a parent node, which is the node containing the element, as well as a sibling defined as the other element contained by the parent node. We refer to the *copath* of an element in a tree as the set comprising its sibling in the tree, the sibling of its parent node in the tree, and so on until reaching the root. An example of a copath is shown in Figure 1.

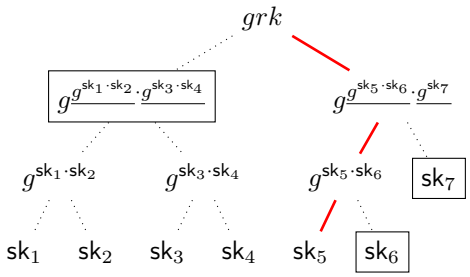


Figure 1: Example computation of a tree key from secret key sk_5. The path from sk_5 to the root is marked in solid red, and the boxed nodes lie on its copath. Thus, an agent who knows sk_5 and the public keys of the boxed nodes can compute grk.

DH groups

We work in a DH group $\mathcal{G}$ for which the decisional DH problem is hard: given a tuple (g^x, g^y, z_b) where $z_0 = g^{xy}$ and $z_1 \leftarrow_{\$} \mathcal{G}$, the advantage of any PPT distinguisher in outputting b is negligible.

We make use of an injection $\iota : \mathcal{G} \to \mathbb{Z}$ mapping group elements to exponents, allowing us to use a group element g^x as the exponent of a new term $g^{\iota(g^x)}$.

As a convention, we use lowercase values k to represent DH secret keys, and uppercase values $K = g^k$ to represent their associated public keys.

Derived Keys

Our protocol contains various different classes of secret and public key, and we fix some names for them here.

Leaf keys λ_j are secret DH keys assigned to tree leaves
Node keys `nk` are secret DH keys assigned to non-leaf tree nodes
Tree keys `tk` are secret values derived at the tree root $T_{0,0}$
Stage keys `sk` are derived by combining the latest `tk` with the previous `sk`, using a hash chain

5 Design

We build on the tree-based group DH schemes described in Section 2.1.4, and give an example tree, path and copath in Figure 1, where the underline denotes the injection from group elements to integers. We give informal explanations of our algorithms in Sections 5.1 and 5.2, as well as precise definitions in pseudocode in Section 5.3.

5.1 Asynchronous Tree Construction

As discussed in Section 2.1.4, distributing the public keys on each agent's copath has normally led to a number of interactive rounds in previous tree DH protocols. We show now that these interactive rounds can be avoided, by using *prekeys* together with a one-time *setup key*.

Prekeys were first introduced by Marlinspike (2013) for asynchronicity in the TextSecure messaging app. They are DH ephemeral values cached by an untrusted intermediate server, and fetched on demand by messaging clients. The prekeys are sent to clients through the public key infrastructure at the same time as long-term identity keys, act as initial messages for a one-round authenticated key exchange protocol, and allow a handshake to take place while only its initiator is online.

We introduce in addition a one-time *setup key*, generated locally by the creator of a group and deleted immediately after use. This key is used to perform an initial key exchange with the prekeys, and allows the initiator to generate secret leaf keys for the other group members while they are offline.

Asynchronous tree construction works as follows. The initiator ("Alice") begins by generating a DH key k_s we call the setup key. She then requests from the public key infrastructure an identity key IK and an ephemeral prekey EK for each of her intended peers ("Bob", "Charlie", ...) numbered 1 through n_{peers}. Using her secret identity key ik_a and the setup key k_s together with the received keys for each peer, she executes a one-round authenticated key exchange protocol to derive leaf keys $\lambda_0, \lambda_1, \ldots, \lambda_{n_{\text{peers}}}$. Using these generated leaf keys, she builds a DH tree whose root holds the initial group key.

We do not force a particular instantiation of this one-round key exchange protocol. Perhaps the simplest instantiation is with an unauthenticated DH exchange between Alice's setup key and Bob's prekey, resulting in an unauthenticated tree structure. This is the design we analyse in Section 6.2. A more practical instantiation is with a strong authenticated key exchange protocol; in Section 6.3 we discuss this version.

To share the initial group key with group member i, Alice sends

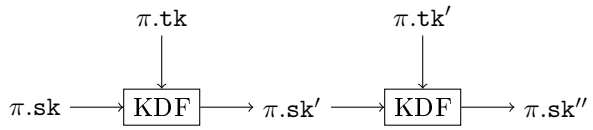


Figure 2: Derivation of stage keys $\pi.\mathtt{sk}$. When a new tree key $\pi.\mathtt{tk}$ is computed (as the root of a DH tree), it is used together with the current stage key to derive a new stage key $\pi.\mathtt{sk}'$, and so on. This "chaining" of keys helps to achieve post-compromise security.

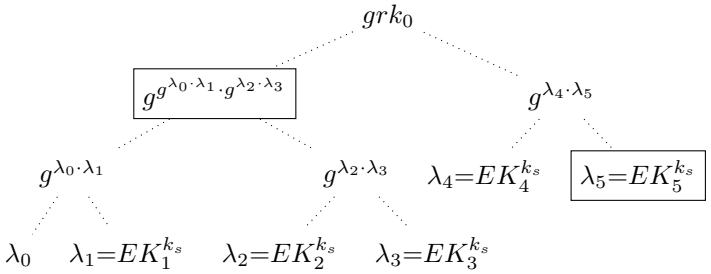


Figure 3: Alice sets up a new tree with herself and five other agents. The copath of Agent 4 is shown boxed.

(i) the public setup key $\mathrm{PK}(k_s)$,
(ii) the public prekey EK_i she used to compute λ_i,
(iii) the public keys along the copath from leaf i to the tree root
(iv) and a MAC of the previous data with i's leaf key.

Upon receiving such a message, each group member can reproduce the computation of the tree key: first they compute their leaf key λ_i, and then they iteratively exponentiate with the public keys on the copath until they reach the final key, which by construction is the tree key $\mathtt{tk}$. They can then compute the stage key $\mathtt{sk}$ from $\mathtt{tk}$ and their previous stage key.

We give a pseudocode definition of this algorithm as Algorithm 1.

5.2 Asynchronous Tree Updates

If Alice's public key *changes*, the data needed by other group members to update their view of the tree—the new (public) DH keys along the path from her leaf node to the tree root—can all be computed by Alice and broadcast at the same time as her new key. Thus, once a tree is constructed, any party can asynchronously generate a new leaf key and broadcast the update to the other group members, who can each update their stored copath as appropriate. This insight allows noninteractive key updates.

Specifically, if at any point Bob wishes to change his leaf key from λ_b to λ_b', he computes the new public keys at all nodes along the path from his leaf to the tree root, and broadcasts to the group his public leaf key together with these public keys. He authenticates this message with a MAC under the previous stage key.

A group member who receives such a message can update their stored copath (at the node on the intersection of the two paths to the root). Computing the key induced by this new path yields the updated group key, again without requiring any two group members to be online at the same time.

5.2.1 Stage key chaining

In order to achieve post-compromise security, stage keys cannot be independent—instead, each stage key must be derived from both the current tree key and the previous stage key. Thus, as long as one of these inputs is unknown to the adversary, the stage key will be as well. The resulting stage keys form a hash chain as depicted in Figure 2.

5.3 Algorithms

We give pseudocode algorithms for all of the operations in our design in Figure 5. As an example, consider the situation where Alice wishes to create a group with five other agents using Algorithm 1. She begins by generating a setup keypair with secret key k_s, and a leaf keypair with secret key λ_0 for herself. She retrieves the public identity and ephemeral prekeys of each of the five agents, and creates the tree shown in Figure 3.

She then sends each agent their respective copath and the prekey she used to set them up in the tree, along with the identities of the other group members and the public setup key. For example, the agent Edward at

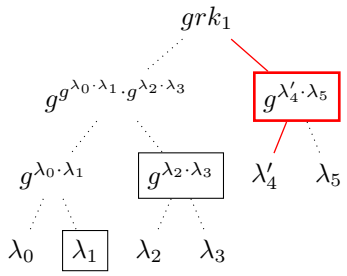


Figure 4: Agent 4 updates their leaf key. The path of Agent 4 is shown in solid red, and the copath of Alice (Agent 0) is shown boxed.

index 4 would receive

$$4, IK_{Alice}, IK_1, \ldots, IK_5, EK_4, K_s, g^{\underline{g^{\lambda_0 \cdot \lambda_1} \cdot g^{\lambda_2 \cdot \lambda_3}}}, g^{\lambda_5}.$$

In her own state, Alice stores her leaf key, the ordered list of public identity keys, the tree key, and her copath. Finally, she derives the stage key used for messaging using Algorithm 6.

Parsing this message (Algorithm 3) allows Edward to identify his position in the group tree, and to construct the group key using Algorithm 2. If Edward then wishes to update his key, he runs Algorithm 4, generating a new leaf key λ'_4 and recomputing the path up to the root. This results in the new tree shown in Figure 4. He sends the key update message

$$4, g^{\lambda'_4}, g^{\underline{g^{\lambda'_4 \cdot \lambda_5}}}$$

comprising his index as well as the path of public keys excluding the root. He stores the updated leaf key and tree key, and computes the new stage key with Algorithm 6.

Upon receiving this key update message, Alice determines her new copath, which has been modified by one of the new public keys sent by Edward. This is done by executing Algorithm 5. From this, she computes the new tree key. Finally, invoking Algorithm 6 computes the new stage key.

6 Security Analysis

We perform our security analysis in two parts.

First, we give a detailed computational security model for multi-stage group key exchange protocols, and instantiate it with an *unauthenticated* version of our construction in which the initial leaf keys are derived directly from the setup key and prekeys. This allows us to capture the core security properties of the key updates, including post-compromise security, without focussing on the properties of the authenticated key exchange used for the initial construction. In the unauthenticated model, we prove indistinguishability of group keys from random values using a game-hopping argument.

Second, we argue that authentication can be easily provided by deriving the initial leaf keys from a non-interactive key exchange, whose security property also applies to the resulting tree key. We give an example construction from the NAXOS protocol (LaMacchia et al., 2007) and verify its authentication property using the TAMARIN prover, modelling the tree construction abstractly as a "black-box" derivation from the leaf keys.

6.1 Computational Model

We build on the multi-stage definition of Fischlin and Günther (2014), in which sessions admit multiple stages with distinct session keys and the adversary can Test any stage, and extend it to group messaging by allowing multiple peers for each session. Our model defines a *security experiment* as a game played between a challenger and a probabilistic, polynomial-time adversary. The adversary is given a set of queries through which it can interact with the challenger, including the ability to relay or modify messages but also to compromise certain secrets. The adversary eventually chooses a so-called Test session, receiving—uniformly at random—either its true session key or a random key sampled from the same distribution. It must then decide which it has received, winning the game if the guess is correct. Thus, a protocol which is secure in this model enjoys the property that an adversary cannot tell if the true keys are replaced with random values.

Definition 1 (Multi-stage key exchange protocol)**.** A multi-stage key exchange protocol Π is defined by a keyspace $\mathcal{K}$, a security parameter λ (dictating the DH group size q) and the following probabilistic algorithms:

(i) KeyGen() $\xmapsto{\$}$ $(\mathsf{pk}, \mathsf{sk})$: generate a DH keypair;

Algorithm 1 Asynchronous group setup

procedure SetupGroup(π, n_{peers})
  $\pi.\lambda_0 \xleftarrow{\$}$ DHKeyGen()
  $k_s \xleftarrow{\$}$ KeyExchangeKeyGen()
  for $i \leftarrow 1 \dots n_{\text{peers}}$ **do** // *generate leaf keys for each agent*
    $IK_i \leftarrow$ public identity key of agent i
    $EK_i \leftarrow$ public ephemeral prekey of agent i
    $\lambda_i \leftarrow \iota\big(\text{KeyExchange}(\pi.ik, IK_i, k_s, EK_i)\big)$
  $T \leftarrow \text{CreateTree}(\lambda_0, \lambda_1, \dots, \lambda_{n_{\text{peers}}})$
  $\pi.ID \leftarrow \text{PK}(\pi.ik), IK_1, \dots, IK_{n_{\text{peers}}}$
  for $i \leftarrow 1 \dots n_{\text{peers}}$ **do**
    $x_i \leftarrow i, \pi.ID, EK_i, \text{PK}(k_s), \text{Copath}(T, i)$
    $m_i \leftarrow$ **message** x_i, $\text{Mac}(x_i; \lambda_i)$ **for** i
  $\pi.\texttt{tk} \leftarrow T_{0,0}$
  $\pi.\bar{P} \leftarrow \text{Copath}(T, 0)$
  return $\pi, m_1, m_2, \dots, m_{n_{\text{peers}}}$

procedure CreateTree($\lambda_0, \lambda_1, \dots, \lambda_n$) // *tree with $n+1$ leaves*
  if $n = 0$ **then return** (leaf, λ_0)
  $l \leftarrow \lceil \log_2(n+1) \rceil - 1$ // *height of the left subtree*
  $(L, lk) \leftarrow \text{CreateTree}(\lambda_0, \dots, \lambda_{(2^l-1)})$ // *complete left subtree*
  $(R, rk) \leftarrow \text{CreateTree}(\lambda_{2^l}, \dots, \lambda_n)$ // *possibly incomplete right subtree*
  $k \leftarrow \iota\big(\text{PK}(lk)^{rk}\big)$
  return $(\text{node}((L, lk), (R, rk)), k)$

procedure Copath(T, i) // *where i is the index of the leaf*
  $l \leftarrow \lceil \log_2 |T| \rceil - 1$ // *height of the left subtree, where $|T|$ is number of leaves*
  if $i < 2^l$ **then** // *i is in the complete left subtree*
    return $\text{PK}(T_{1,1}) \parallel \text{Copath}(T_{1,0}, i)$
  else// *i is in the possibly incomplete right subtree*
    return $\text{PK}(T_{1,0}) \parallel \text{Copath}(T_{1,1}, i - 2^l)$

Algorithm 2 Deriving keys on path to root

procedure PathNodeKeys($\lambda, \bar{P}$) // *leaf key and the copath of public keys*
  $\texttt{nks}_{|\bar{P}|} \leftarrow \lambda$
  for $n \leftarrow (|\bar{P}| - 1) \dots 1$ **do**
    $\texttt{nks}_n \leftarrow \iota\big((\bar{P}_n)^{\texttt{nks}_{n+1}}\big)$
  return $(\bar{P}_0)^{\texttt{nks}_1}, \texttt{nks}_1, \dots, \texttt{nks}_{|\bar{P}|}$

Algorithm 3 Receiving a setup message as agent at index i

procedure ProcessSetupMessage(π)
  receive $i, ID, EK, K_s, \bar{P}$
  $(\pi.ID, \pi.\bar{P}) \leftarrow (ID, \bar{P})$ // *store agent ids and copath in state*
  $ek \leftarrow$ ephemeral prekey corresponding to EK from π
  $\pi.\lambda \leftarrow \iota\big(\text{KeyExchange}(\pi.ik, ID_0, ek, K_s)\big)$ // *compute initial leaf key*
  $\texttt{nks} \leftarrow \text{PathNodeKeys}(\pi.\lambda, \pi.\bar{P})$
  $\pi.\texttt{tk} \leftarrow \texttt{nks}_0$ // *store initial tree key*
  return π

Algorithm 4 Updating a key as agent at index i

procedure UpdateKey(π)
  $\pi.\lambda \xleftarrow{\$}$ DHKeyGen()
  $\texttt{nks} \leftarrow \text{PathNodeKeys}(\lambda, \pi.\bar{P})$
  $x \leftarrow i, \text{PK}(\texttt{nks}_1), \dots, \text{PK}(\texttt{nks}_{|\bar{P}|})$
  $m \leftarrow$ **message** x, $\text{Mac}(x; \pi.sk)$ **for all**
  $\pi.\texttt{tk} \leftarrow \texttt{nks}_0$
  return π, m

Algorithm 5 Processing a key update message as agent at index i

procedure ProcessUpdateMessage(π)
  receive j, U // *their index and public keys along their path*
  $h \leftarrow \text{IndexToUpdate}(\lceil \log_2 |ID| \rceil, i, j)$
  $\pi.\bar{P}_h \leftarrow U_h$ // *index h of the copath has been updated in this message*
  $\texttt{nks} \leftarrow \text{PathNodeKeys}(\pi.\lambda, \pi.\bar{P})$
  $\pi.\texttt{tk} \leftarrow \texttt{nks}_0$
  return π

procedure IndexToUpdate(h, i, j)
  if $(i < 2^{h-1}) \wedge (j < 2^{h-1})$ **then** // *both are in the left subtree*
    return $\text{IndexToUpdate}(h - 1, i, j)$
  else if $(i \geq 2^{h-1}) \wedge (j \geq 2^{h-1})$ **then** // *both in the right subtree*
    return $\text{IndexToUpdate}(h-1, i-2^{h-1}, j-2^{h-1})$
  return h // *otherwise return index where they differ*

Algorithm 6 Deriving the stage key

procedure DeriveStageKey(π)
  $\pi.\texttt{sk} \leftarrow \text{KDF}(\pi.\texttt{sk}, \pi.\texttt{tk}, \pi.ID)$
  return π

Figure 5: Pseudocode descriptions of the algorithms in our design. Informal explanations can be found in Section 5.

Table 2: Adversary queries defined in our model. We use u to denote the agent targeted by a query, i to denote the index of a session at an agent, and t to denote the stage of a session—thus, for example, $(Alice, 3, 4)$ would denote the fourth stage of Alice's third session. We use m for messages and b for a random bit.

$\mathsf{Create}(u, v_1, v_2, \ldots v_{n-1})$	Given a set of intended peers $v_1, \ldots, v_{n-1}$ $(n \leq \gamma)$, agent u executes Activate and returns its output as the initial message. This query models creating a new session.
$\mathsf{Send}(u, i, m)$	Given a session (u, i) and a message, execute Run with m and the current state of the session, updating its state and returning the resulting message. This query models sending a message to a session.
$\mathsf{RevSessKey}(u, i, t)$	Given (u, i, t), return the session key generated in that stage, if it exists. This query models session keys being leaked to the adversary and is used to capture authentication properties.
$\mathsf{RevRandom}(u, i, t)$	Given (u, i, t), reveal the random coins by u in stage t of session s. This query models the corruption of an agent, either in their initial key generation (at stage $t = 0$) or in later stages $(t > 0)$.
$\mathsf{Test}(u, i, t)$	Given (u, i, t), let k_0 denote the session computed by user u at stage t of session i, and let k_1 denote a uniformly randomly sampled key from the challenger. The challenger flips a coin $b \leftarrow_\$ \{0, 1\}$ and returns k_b.
$\mathsf{Guess}(b')$	The adversary immediately terminates its execution after this query.

(ii) Activate$(\mathsf{sk}, \rho, peers) \to \pi$: initialise an agent's protocol state by accepting a long-term secret key sk, a role ρ and a list *peers* of peers and returning a state π; and
(iii) Run$(\pi[, m]) \to (\pi'[, m'])$: starting from some protocol state π and an optional incoming message m, execute the protocol to derive an updated state π' and an optional outgoing message m'.

Protocols may maintain state between sessions—indeed, Cohn-Gordon et al. (2016) prove that this is necessary for post-compromise security—and we collect this state π into the following variables.

Definition 2 (State)**.** A state π is a collection of the following variables:

(i) $\pi.stage$, the current stage t of the session (initialised to 0 and incremented after each new stage session key is computed)
(ii) $\pi.\mathtt{sk}$, the agent's secret stage key to be used at the current stage.
(iii) $\pi.\mathtt{active}[s]$, the execution status for stage s. Takes the value `active` at the start of a session, and later set to either `accept` or `reject` when the session key is computed
(iv) $\pi.\mathsf{k}[s]$, the session key output by stage s.
(v) $\pi.\mathtt{tk}$, the tree key of the current stage.
(vi) $\pi.\lambda$, the leaf key of the current stage.

Definition 3 (Adversary queries)**.** We allow the adversary access to the queries defined in Table 2.

We fix a maximum group size γ, which is the largest group that an agent is willing to create. This can be application-specific.

6.2 Unauthenticated Computational Analysis

We can now analyse our protocol in the model of Section 6.1. In this analysis we do not consider the use of long-term keys, considering them instead as used in the first stage. Our freshness criteria allow the adversary to corrupt the random values or session key from any stage, but rule out trivial attacks created by such corruptions.

We define Activate to additionally create locations in the state for the agent's belief about the shape of a tree in a given session:

Definition 4 (Tree structure)**.** Let π additionally contain

(i) $\pi.\bar{P}$, the copath of the agent
(ii) $\pi.ID$, an ordered list of identifiers for peers in the group, where the index of each identifier is the index of that peer's leaf.
(iii) optional data associated to each peer;
(iv) optional DH public keys attached to all nodes in the tree;

We define Run with the algorithms from Section 5.3, with

$$\textsc{KeyExchange}(\pi.ik, ID_0, ek, K_s) := {K_s}^{ek}.$$

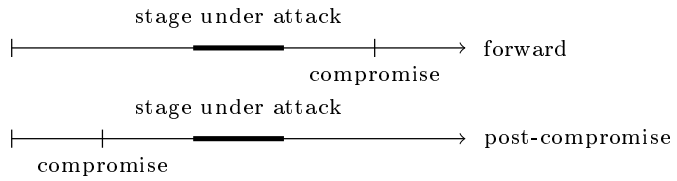


Figure 6: Attack scenarios of forward and post-compromise secrecy, with the session under attack marked in **bold**. Forward secrecy protects sessions against later compromise; post-compromise secrecy protects sessions against earlier compromise.

That is, our initial leaf nodes are constructed unauthenticated from initial ephemeral keys. In this setting we do not need the MACs which are defined in the protocol algorithms, and we do not make any assumptions here on their security properties.

Thus armed, we can define when two agents are intended communication partners. We use a matching-conversations definition:

Definition 5 (Matching)**.** We say that two stages (u, i, t) and (v, j, s) *match* if they both have status `accept` and moreover have derived the same session key.

Definition 6 (Freshness of a copath)**.** Let $\bar{P} = \bar{P}_0, \ldots, \bar{P}_{|\bar{P}|-1}$ be a list of group elements representing a copath and let $\ell = \lambda_0 \ldots \lambda_{n-1}$ be a list of group elements representing leaf keys. We say that $\bar{P}$ is the i^{th} *copath induced by* ℓ precisely if, in the DH tree induced by ℓ, each $\bar{P}_j$ is the sibling of a node on the path from λ_i to the tree root.

We say that a copath $\bar{P}$ is *fresh* if both

(i) $\bar{P}$ is the i^{th} copath induced by some ℓ, and

(ii) for each $g^{\lambda_j} \in \ell$, both

  (a) λ_j was returned by a Send or Create query to some stage (u, i, t), and

  (b) no RevRandom(u', i', t') query was issued.

Intuitively, a copath is fresh if it is built from honestly-generated and unrevealed leaf keys. In particular, the copath's owner's leaf key must also be unrevealed, since it is included in ℓ.

Definition 7 (Freshness of a stage)**.** We say that a stage (u, i, t) deriving session key *sessk* is *fresh* if all of the following hold:

(i) it has status `accept`,

(ii) the adversary has not issued a RevSessKey(u, i, t) query,

(iii) there does not exist (v, j, s) such that the adversary has issued a query RevSessKey(v, j, s) whose return value is *sessk*, and

(iv) one of the following criteria holds:

  (a) $t > 0$ and stage $(u, i, t-1)$ is fresh, or

  (b) the current copath is fresh.

Intuitively, a stage is fresh if *either* all of the leaves in the current tree are fresh *or* the previous stage was fresh. The latter disjunct captures a form of post-compromise security: if an adversary allows a fresh stage to `accept`, subsequent stages will also be fresh.

Capturing strong security properties Our notion of stage freshness captures the strong security properties discussed in Section 3, by allowing the adversary to Test sessions under a number of compromise scenarios. Specifically:

authentication states that if the ephemeral keys used in a session are from an uncorrupted session then only the agents who generated them can derive the group key. Indeed, for a stage to be fresh either it or one of its ancestors must have had a fresh copath; that is, one that is built only from λ_j which were sent by other honest stages.

PFS is captured through clause (iv)b and the definition of the RevRandom query. Indeed, suppose Alice accepts a stage t and then updates her key in stage $t+1$. An adversary who queries RevRandom$(\ldots, t+1)$ does not receive the randomness from stage t, which therefore remains fresh. Our model thus requires the key of stage t to be indistinguishable from random to such an adversary.

PCS is captured through clause (iv)a. Indeed, suppose the adversary has issued RevRandom queries against all of Alice's stages from 0 to t *except* some stage $0 \leq j < t$. Absent other queries, stage j is therefore considered fresh, and hence by clause (iv)a stages $j+1, j+2, \ldots, t$ are fresh as well. Our model thus requires their keys to be indistinguishable from random to such an adversary.

Definition 8 (Security experiment)**.** At the start of the game, the challenger generates the public/private key pairs of all $\mathsf{n_P}$ parties and sends all public info including the identities and public keys to the adversary. The adversary then asks a series of queries before eventually issuing a $\mathsf{Test}(u, i, t)$ query, for the t^{th} stage of the i^{th} session of user u. We can equivalently think of the adversary as querying oracle machines π_u^i for the i^{th} session of user u.

Our notion of security is that the group session key of the Tested session stage is indistinguishable from random. Thus, after the $\mathsf{Test}(u, i, t)$ query, the challenger flips a coin $b \leftarrow_{\$} \{0, 1\}$ and with probability $1/2$ (when $b = 0$) reveals the actual session key of user u's ith session at stage t to the adversary, and with probability $1/2$ (when $b = 1$) reveals a uniformly randomly chosen key instead. The adversary is allowed to continue asking queries. Eventually the adversary must guess the bit b with a $\mathsf{Guess}(b')$ query before terminating. If the Tested (u, i, t) satisfies fresh and the guess is correct ($b = b'$), the adversary wins the game. Otherwise, the adversary loses.

We say that a multi-stage key exchange protocol is *secure* if the probability that any probabilistic polynomial-time adversary wins this game is bounded above by $1/2 + \mathsf{negl}(\lambda)$, where $\mathsf{negl}(\lambda)$ tends to zero faster than any polynomial in the security parameter λ.

We now give our theorem and sketch the proof. The full proof appears in the appendix.

Theorem 6.1. *Let $\mathsf{n_P}$, $\mathsf{n_S}$ and $\mathsf{n_s}$ denote bounds on the number of parties, sessions and stages in the security experiment respectively. Under the decisional DH assumption, where ι is instantiated as a random oracle, the success probability of any ppt adversary against the key indistinguishability game of our protocol is bounded above by*

$$1 - \prod_{i=0}^{\mathsf{n_P}\mathsf{n_S}\mathsf{n_s}-1} (1 - i/q) + (\mathsf{n_S}\mathsf{n_s})^{\gamma}\mathsf{n_P}^{\gamma+1}\left(\epsilon_{DDH} + 1/q\right) + \mathsf{negl}(\lambda)\,,$$

where ϵ_{DDH} bounds the advantage of a PPT adversary against the decisional DH game.

Proof Sketch. Our proof uses the standard game hopping technique. We start at our original security game and consider ("hop to") similar games, bounding the success probability of the adversary in each hop, until we reach a game that the adversary clearly cannot win with a probability non-negligibly over $1/2$. As all the games' probabilities are bounded to one another, we are able to bound the overall success probability of the adversary in the original security game.

The overall structure of the proof is as follows. First, we perform some administrative game hops to rule out the possibility of DH key collisions. Then, we guess the indices (u, i, t) of the Test session and stage. If it is not fresh then the adversary does not win. If it is fresh, we perform a case distinction based on the condition of the freshness predicate which it satisfies: either the current copath is fresh or a previous stage was fresh.

In the latter case, indistinguishability holds by induction. In the former case, by definition we know that all of the leaf keys used to generate the current stage are honestly-generated and unrevealed. The secret key at a node with child public keys g^x and g^y is defined to be g^{xy}, and thus by hardness of the decisional DH problem (DDH) we can indistinguishably replace it with a random group element. We perform this replacement in turn for each non-leaf node in the tree, bounding the probability difference at each game hop with the DDH advantage. After all non-leaves have been replaced, the tree key (and hence the stage session key) is replaced with a random group element. The success probability of the adversary against this final game is therefore no better than $1/2$. By summing probabilities throughout the various cases we derive our overall probability bound. □

6.3 Adding Authentication

Deriving the leaf keys λ_j from a one-round authenticated key exchange protocol allows for authentication of the initial group key, in the sense that only an agent who can complete the key exchange protocol can derive the group key. We now give an example of such a construction, and analyse its authentication property using the TAMARIN prover.

We use NAXOS (LaMacchia et al., 2007) as our one-round key exchange protocol. In NAXOS, an agent A sends a message $X = g^x$ where $x = H_1(esk_A, sk_A)$, receives a message Y, and derives a shared key $K = H_2(Y^{sk_A}, pk_B^x, Y^x, A, B)$. Schmidt et al. (2012) performed an automated analysis of NAXOS in a symbolic version of the eCK model.

To model the authentication property we abstract out the tree construction and replace it with a symbolic "oracle" which assigns to any set of public keys a fresh term representing the group key they induce. The adversary or any agent is permitted to query this oracle, with the restriction that they must prove knowledge of at least one secret key corresponding to a public key in the set.

We use TAMARIN to verify this construction. Roughly, we model a protocol role Alice who accepts initial NAXOS messages, representing new group members, and adds the resulting session keys to her state. At any point she may stop accepting new members and instead derive a group key by querying our abstract oracle.

We remark that although using a more advanced authenticated key exchange protocol for the leaves is a relatively small change, the resulting security property does not follow trivially. In an earlier design, we considered a protocol without authentication of the initial messages. We analysed this earlier design and TAMARIN found an attack in which Alice correctly fetches prekeys, computes a group key and sends the resulting (abstract) copath to Bob, but the adversary modifies this message to add a malicious leaf key. Knowing a leaf key for Bob's tree, it can then derive the resulting session key even though it is accepted by Bob. The TAMARIN analysis made it clear that for the group key to be authenticated, not just the λ_j but also the copath of public keys needs to be authenticated, and we improved our design accordingly.

We will release the TAMARIN models shortly. The model verifies that the initial group key an agent derives in a group of size three is secret, if none of the agents they believe to be in the group have been compromised. Although TAMARIN supports unbounded verification, and we consider an arbitrary number of parties and instances, it was necessary to bound the verified security property to group sizes of three for verification to terminate quickly. The verification of this security property proceeds automatically using three helper lemmas and takes a few minutes on a modern desktop.

7 Proof-of-Concept Implementation

We implemented the protocol described in Section 5 in Java as a proof of concept. Our implementation generates public keys for a number of users, constructs a tree as described in Figure 5, and encrypts and decrypts messages. We implement the tree as a recursive data structure, and perform updates recursively as opposed to iteratively.

Message passing We pass messages between sessions as native Java objects in memory, to avoid (de)serialisation overhead in our benchmarks. For simplicity, instead of extracting and sending separate copaths to each group member we simply distribute the entire public tree as a Java object, from which each node extracts its own copath.

Authentication Again for simplicity, we instantiate the protocol with the unauthenticated design of Section 6.2, referring to the initial keys as identity keys. We remark that using a one-round key exchange protocol instead would simply require modifying the `newSession` and `fromSessionInitiationMessage` methods, alongside the routines to serialise to message objects, and the key generation in the benchmarking code.

Primitives For our Diffie-Hellman group operations we used a Java implementation (Skiba, 2008) of Curve25519 (Bernstein, 2006). Encryption and decryption of messages uses Java's native AES-GCM support, at 128 bits to allow running the example without the Java runtime patch necessary for 256 bit keys.

Encryption keys for messages ("message keys") are derived similarly to the approach used in the Double Ratchet algorithm: chain keys are derived from a stage key, and then message keys from the chain keys. We also extend the described algorithm to salt the session root when deriving it from the root of the tree, to ensure that even if multiple trees were generated with the same public keys at their leaves, they would derive a different root key. In our example we use HKDF for all key derivations of both stage keys and message keys.

Effort While a production-ready implementation may take significantly more work to build, our toy example took took a few days to build and totalled 1480 lines of code, demonstrating that this is not an overly complex set of algorithms to implement.

Our goal was not to produce secure cryptographic code, but to demonstrate the feasibility of our algorithms for practical scenarios. Even in unoptimised Java, it takes fractions of a second to import a large tree—as expected, given our logarithmic asymptotic complexity. In Table 3 we give some simple timing results for group construction and key update messages for variously-sized groups.

Our asynchronous setup requires the initiator to construct their entire key tree locally, in order to generate the public tree values to send to other group members. Thus, for the group creator it takes linear time in the size of the group. Although this overhead is minimal for the group sizes normally seen in messaging applications, for large-scale use cases we remark that this is a significant performance requirement. However, we might expect large-scale use cases either to be more tolerant of a high setup cost, or to build up to a large size gradually by dynamically adding group members.

Table 3: Times in milliseconds for our implementation to perform various tree operations. All computation was performed on a 2016 Apple MacBook Pro, and results are the average of 5 benchmark runs. In *construct tree*, the initiator fetches public keys for each group member, and follows the unauthenticated algorithm in Section 5 to build a complete DH tree. In *import tree*, responders use their private key to compute the first stage key. In encrypt (initiator), the initiator derives new chain keys from the first stage key, and uses them to encrypt messages, decrypted by the responder in *decrypt (responder)*. In *encrypt (responder)*, the responder performs a tree update as per Section 5 to derive a new stage key and then message key, encrypting a message decrypted in *decrypt (initiator)*.

group size	construct tree (initiator)	import tree (responder)	encrypt (initiator)	decrypt (responder)	encrypt (responder)	decrypt (initiator)
2	2	0	0	1	0	0
7	6	0	0	1	0	0
127	91	0	0	2	0	0
32,768	25,188	21	1	4	0	0
100,000	77,023	249	1	4	0	0

8 Extensions

We here remark on various possible extensions to our designs. In general, because we use standard tree-DH techniques, much of the existing literature is directly applicable. This means that we can directly apply well-studied techniques which do not require interactive communication rounds.

Sender-specific authentication As early as 1999, Wallner et al. (1999) pointed out the issue of "sender-specific authentication": in a system which derives a shared group key known to all members, there is no cryptographic proof of *which* group member sent a particular message. Various works have discussed such proofs; the most common design is to assign to each group member a signature key with which they sign all their messages. We remark that it is easy to extend our design with such a system.

In particular, if an agent generates and broadcasts new signature keys together with their new leaf keys, signing the new key with the old one, then we conjecture that they will achieve a form of authentication even post-compromise.

Dynamic groups We refer the reader to e.g. (Kim et al., 2001) for a summary of previous work on dynamic groups. In general, since we build on tree-based ideas, our design can support join and leave operations using standard techniques.

We remark in particular that these operations can be done *asynchronously* using a design similar to the setup keys in Section 5.1. Specifically, Alice can add Ted as a sibling to her own node in the tree by performing an operation similar to the initial tree setup, generating an ephemeral key and performing a key update which replaces Alice's leaf with an intermediate node whose children are Alice and Ted. With the cooperation of other users in the tree, Alice can add Ted *anywhere*, allowing her to keep the tree balanced.

Multiple Devices One important motivation for supporting group messaging is to enable users to communicate using more than one of their own devices. By treating each device as a separate group member, our design of course supports this use case. However, the tree structure can be optimised for this particular scenario: all of Alice's devices can be stored in a single subtree, so that the "leaves" of the group tree are themselves roots of device-specific trees. This has two particular benefits.

First, in her own time Alice can execute the group key agreement protocol just between her devices, and use the resulting shared secret as an ephemeral prekey or "pretree". This allows other group members to retrieve a single key for Alice, instead of adding all of her devices as separate entities in the group tree. If most users have multiple devices, this can significantly reduce the size of group trees.

Second, when any of Alice's devices performs a key update, the other group members only need to know the public keys from the root of Alice's subtree to the root of the group tree. In particular, Alice does not need to broadcast to the group the set of her devices or the metadata about which device is performing a key update. Thus, she can maintain end-to-end encryption between all of her devices while still keeping the list private.

Chain keys The Signal protocol introduced the concept of *chain keys* to support out-of-order message receipt as well as a fine-grained form of forward secrecy. Instead of using a shared secret to encrypt messages directly, Signal derives a new encryption key for each message by applying a key derivation function to the current key, generating a new chain key in the process.

The shared secret derived by our group key exchange can be directly used as the start of a key chain. Indeed, our implementation derives its message keys from a hash chain, ensuring that each key is only ever used once as well as providing a form of forward secrecy after compromise of a chain key.

9 Conclusion

While modern messaging applications can offer strong security guarantees, they typically only do this for two-party communications. If another person is added to the group, the effective security guarantees are decreased, without notifying the users of this security degradation.

In this paper, we combined techniques from synchronous group messaging with modern guarantees from asynchronous messaging. Our resulting asynchronous design combines the bandwidth benefits of group messaging with the strong security guarantees of modern point-to-point protocols. This paves the way for modern messaging applications to offer the same type of security for groups that they are currently only offering for two-party communications.

Our construction is of independent interest, since it provides a blueprint for generically applying insights from synchronous group messaging in the asynchronous setting. We expect this to lead to many more alternative designs in future works.

References

Michel Abdalla, Céline Chevalier, Mark Manulis, and David Pointcheval. 2010. Flexible Group Key Exchange with On-demand Computation of Subgroup Keys. In *AFRICACRYPT 10 (LNCS)*, Daniel J. Bernstein and Tanja Lange (Eds.), Vol. 6055. Springer, Heidelberg, 351–368.

Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *PKC 2006 (LNCS)*, Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin (Eds.), Vol. 3958. Springer, Heidelberg, 207–228.

Dan Boneh and Alice Silverberg. 2003. Applications of multilinear forms to cryptography. In *Topics in Algebraic and Noncommutative Geometry: Proceedings in Memory of Ruth Michler*, Caroline Grant Mellesand Jean-Paul Brasseletand Gary Kennedyand Kristin Lauter and Lee McEwan (Eds.). Contemporary Mathematics, Vol. 324. American Mathematical Society.

Dan Boneh and Mark Zhandry. 2014. Multiparty Key Exchange, Efficient Traitor Tracing, and More from Indistinguishability Obfuscation. In *CRYPTO 2014, Part I (LNCS)*, Juan A. Garay and Rosario Gennaro (Eds.), Vol. 8616. Springer, Heidelberg, 480–499. https://doi.org/10.1007/978-3-662-44371-2_27

Nikita Borisov, Ian Goldberg, and Eric Brewer. 2004. Off-the-record Communication, or, Why Not to Use PGP. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society (WPES '04)*. ACM. https://doi.org/10.1145/1029179.1029200

Timo Brecher, Emmanuel Bresson, and Mark Manulis. 2009. Fully Robust Tree-Diffie-Hellman Group Key Exchange. In *CANS 09 (LNCS)*, Juan A. Garay, Atsuko Miyaji, and Akira Otsuka (Eds.), Vol. 5888. Springer, Heidelberg, 478–497.

Emmanuel Bresson, Olivier Chevassut, David Pointcheval, and Jean-Jacques Quisquater. 2001. Provably Authenticated Group Diffie-Hellman Key Exchange. In *ACM CCS 01*. ACM Press, 255–264.

Yi-Ruei Chen and Wen-Guey Tzeng. 2017. Group key management with efficient rekey mechanism: A Semi-Stateful approach for out-of-Synchronized members. *Computer Communications* 98 (2017). https://doi.org/10.1016/j.comcom.2016.08.001

Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. 2016. On post-compromise security. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*. IEEE, 164–178.

Cas J. F. Cremers and Michele Feltz. 2012. Beyond eCK: Perfect Forward Secrecy under Actor Compromise and Ephemeral-Key Reveal. In *ESORICS 2012 (LNCS)*, Sara Foresti, Moti Yung, and Fabio Martinelli (Eds.), Vol. 7459. Springer, Heidelberg, 734–751.

Yvo Desmedt, Tanja Lange, and Mike Burmester. 2007. Scalable Authenticated Tree Based Group Key Exchange for Ad-Hoc Groups. In *FC 2007 (LNCS)*, Sven Dietrich and Rachna Dhamija (Eds.), Vol. 4886. Springer, Heidelberg, 104–118.

eQualit.ie. 2016. $(N+1)$SEC. (2016). https://learn.equalit.ie/wiki/Np1sec

Facebook. 2017. *Messenger Secret Conversations (Technical Whitepaper Version 2.0)*. Technical Report. https://fbnewsroomus.files.wordpress.com/2016/07/messenger-secret-conversations-technical-whitepaper.pdf

Marc Fischlin and Felix Günther. 2014. Multi-stage key exchange and the case of Google's QUIC protocol. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1193–1204.

Ian Goldberg, Berkant Ustaoglu, Matthew Van Gundy, and Hao Chen. 2009. Multi-party off-the-record messaging. In *ACM CCS 09*, Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis (Eds.). ACM Press, 358–368.

Matthew D. Green and Ian Miers. 2015. Forward Secure Asynchronous Messaging from Puncturable Encryption. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 305–320. https://doi.org/10.1109/SP.2015.26

The Guardian. 2017. Contact the Guardian securely. (2017). https://gu.com/tip-us-off

internet.org. 2016. State of Connectivity 2015. (2016). https://fbnewsroomus.files.wordpress.com/2016/02/state-of-connectivity-2015-2016-02-21-final.pdf

Antoine Joux. 2004. A One Round Protocol for Tripartite Diffie-Hellman. *Journal of Cryptology* 17, 4 (Sept. 2004), 263–276.

Yongdae Kim, Adrian Perrig, and Gene Tsudik. 2000. Simple and Fault-tolerant Key Agreement for Dynamic Collaborative Groups. In *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS '00)*. ACM. https://doi.org/10.1145/352600.352638

Yongdae Kim, Adrian Perrig, and Gene Tsudik. 2001. *Communication-Efficient Group Key Agreement*. Springer US. https://doi.org/10.1007/0-306-46998-7_16

Yongdae Kim, Adrian Perrig, and Gene Tsudik. 2004. Tree-based Group Key Agreement. *ACM Trans. Inf. Syst. Secur.* (Feb. 2004). https://doi.org/10.1145/984334.984337

N. Kobeissi, K. Bhargavan, and B. Blanchet. 2017. Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. to appear.

Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. 2007. Stronger Security of Authenticated Key Exchange. In *ProvSec 2007 (LNCS)*, Willy Susilo, Joseph K. Liu, and Yi Mu (Eds.), Vol. 4784. Springer, Heidelberg, 1–16.

Sangwon Lee, Yongdae Kim, Kwangjo Kim, and Dae-Hyun Ryu. 2003. An Efficient Tree-Based Group Key Agreement Using Bilinear Map. In *ACNS 03 (LNCS)*, Jianying Zhou, Moti Yung, and Yongfei Han (Eds.), Vol. 2846. Springer, Heidelberg, 357–371.

Fermi Ma and Mark Zhandry. 2017. Encryptor Combiners: A Unified Approach to Multiparty NIKE, (H)IBE, and Broadcast Encryption. Cryptology ePrint Archive, Report 2017/152. (2017). http://eprint.iacr.org/2017/152.

Moxie Marlinspike. 2013. Forward Secrecy for Asynchronous Messages. (22 08 2013). https://whispersystems.org/blog/asynchronous-security/

Moxie Marlinspike. 2016. Signal Protocol documentation. (2016). https://whispersystems.org/docs/

Benedikt Schmidt, Simon Meier, Cas J. F. Cremers, and David A. Basin. 2012. Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. 78–94. https://doi.org/10.1109/CSF.2012.25

Victor Shoup. 2004. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptology EPrint Archive* 2004 (2004), 332.

Dmitry Skiba. 2008. trevorbernard/curve25519-java. (23 02 2008). https://github.com/trevorbernard/curve25519-java

D. G. Steer, L. Strawczynski, W. Diffie, and M. Wiener. 1990. *A Secure Audio Teleconference System*. Springer New York. https://doi.org/10.1007/0-387-34799-2_37

D. Wallner, E. Harder, and R. Agee. 1999. Key Management for Multicast: Issues and Architectures. (1999).

WhatsApp. 2016. *WhatsApp Encryption Overview*. Technical Report. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf

Chung Kei Wong, Mohamed Gouda, and Simon S. Lam. 2000. Secure Group Communications Using Key Graphs. *IEEE/ACM Transactions on Networking* 8, 1 (Feb. 2000), 16–30.

Zheng Yang, Chao Liu, Wanping Liu, Daigu Zhang, and Song Luo. 2017. A new strong security model for stateful authenticated group key exchange. *International Journal of Information Security* (2017), 1–18. https://doi.org/10.1007/s10207-017-0373-1

A Computational Security Proof

Theorem 6.1. *Let* $\mathsf{n_P}$, $\mathsf{n_S}$ *and* $\mathsf{n_s}$ *denote bounds on the number of parties, sessions and stages in the security experiment respectively. Under the decisional DH assumption, where* ι *is instantiated as a random oracle, the success probability of any ppt adversary against the key indistinguishability game of our protocol is bounded above by*

$$1 - \prod_{i=0}^{\mathsf{n_P n_S n_s}-1} (1 - i/q) + (\mathsf{n_S n_s})^{\gamma} \mathsf{n_P}^{\gamma+1} (\epsilon_{DDH} + 1/q) + \mathsf{negl}(\lambda)\,,$$

where ϵ_{DDH} *bounds the advantage of a PPT adversary against the decisional DH game.*

Proof. Security in this sense means that no efficient adversary can break the key indistinguishability game against our protocol. Suppose for contradiction that $\mathcal{A}$ is such an adversary. By the definition of the security experiment, it can only win if it issues a $\mathsf{Test}(u, i, t)$ query against some stage t of a session i at agent u such that (u, i, t) is fresh, and subsequently issues a correct $\mathsf{Guess}(b)$ query.

By the definition of freshness, (u, i, t) is fresh exactly when

(i) it has status `accept`,
(ii) the adversary has not issued a $\mathsf{RevSessKey}(u, i, t)$ query,
(iii) there does not exist (v, j, s) such that the adversary has issued a query $\mathsf{RevSessKey}(v, j, s)$ whose return value is $sessk$, and
(iv) one of the following criteria holds:
  (a) $t > 0$ and stage $(u, i, t-1)$ is fresh, or
  (b) the current copath is fresh.

We proceed by constructing a sequence of related games and corresponding adversaries, identifying the game from the security experiment with game 0. Let Adv_i denote the maximum over all adversaries $\mathcal{A}$ of the advantage of $\mathcal{A}$ in game i. Our goal is to bound Adv_0, the advantage of any adversary against the security experiment.

Game 0. This is the original AKE security game. We see that the success probability of the adversary is bounded above by

$$1/2 + \mathsf{Adv}_0$$

Game 1. This is the same as Game 0, except the challenger aborts and the adversary loses if there is ever a collision of honestly generated DH keys in the game. There are a total number of $\mathsf{n_P}$ parties in the game. There are a maximum of $\mathsf{n_S n_s}$ ephemeral DH keys generated per party. There are therefore a total maximum of $\mathsf{n_P n_S n_s}$ DH keys, each pair of which must not collide. All keys are generated in the same DH group of order q. Therefore we have the following bound:

$$\mathsf{Adv}_0 \leq 1 - \prod_{i=0}^{\mathsf{n_P n_S n_s}-1} (1 - i/q) + \mathsf{Adv}_1$$

Game 2. This is the same as Game 1, except the challenger begins by guessing (uniformly at random, independently of other random samples) a user u', session i' and stage t'. If the adversary issues a $\mathsf{Test}(u, i, t)$ query with $(u, i, t) \neq (u', i', t')$, the challenger immediately aborts the game and the adversary loses.

Since the challenger's guess is independent of the adversary's choice of Test session, we derive the bound

$$\mathsf{Adv}_1 \leq \mathsf{n_P n_S n_s} \cdot \mathsf{Adv}_2$$

Now, note that the Test session has an internal view of the DH tree structure and the peers associated with each leaf.

Game 3. In this game, the challenger guesses in advance the peer sessions associated with each of these leafs. There are of course no more than $\gamma - 1$ leafs, so we will assume the worst case of making $\gamma - 1$ guesses. In other words, this game hop is guessing the (ordered) peers in the Test.

Precisely, the challenger does the following. For each leaf l, it guesses a triple of indices $(v'_l, j'_l, s'_l) \in [n_{\mathsf{n_P}}] \times [\mathsf{n_S}] \times [\mathsf{n_s}]$ and aborts if at leaf l there exists a session π_v^j that matches the Test session π_u^i at stage s but $(v'_l, j'_l, s'_l) \neq (v, j, s)$. Note that it might be the case that no such matching π_v^j exists, but this game ensures that if such π_v^j do exist, they are uniquely defined and known in advance by the challenger.

We must first show that there can exist at most one (v, j, s) that matches the Test that also has the same internal view of being at any given leaf node of the tree structure of the Test. This follows from our first game hop that forced honestly generated DH keys not to collide and the fact that sessions matching includes computing the same session key, and the ordered list of identities with keys is in the KDF.

This provides the following bound.

$$\mathsf{Adv}_2 \leq (\mathsf{n_P n_S n_s})^{\gamma-1} \cdot \mathsf{Adv}_3$$

Consider the event E defined to be true when the copath in the session state of the Tested stage is fresh. We now perform a case distinction on E, considering first the case (i) where E is true, and then the case (ii) where E is false.

Case (i). We assume that E holds. By definition of copath freshness, it therefore holds that the copath is the i^{th} copath induced by some ℓ, where each $\lambda_j \in \ell$ was output by an honest stage against which no RevRandom query was issued. WLOG we define λ_1 to be the actor of the Test session's DH leaf key.

Case (i), Game $4(i).1$.

Recall that the parent of the first two leaf nodes, λ_1 and λ_2, is defined as $g^{\lambda_1 \cdot \lambda_2}$. We define a new game in which, in the local session key computation of the actor of the Test session and any match (which is unique by the previous game), $g^{\lambda_1 \cdot \lambda_2}$ is replaced with a group element g^z sampled uniformly at random, and all subsequent computations upwards along the path of the tree use g^z instead of $g^{\lambda_1 \cdot \lambda_2}$.

This is a game hop based on indistinguishability (Shoup, 2004). In general, we consider a hybrid game and a distinguisher $\mathcal{D}$ that interpolates between the two games. The distinguisher $\mathcal{D}$ that distinguishes between distributions P_1 and P_2, when given an element drawn from distribution P_1 as input, outputs 1 with probability $\mathsf{Adv}_3 + 1/2$, and when given element drawn from distribution P_2, outputs 1 with probability $\mathsf{Adv}_{4(i).1} + 1/2$. The indistinguishability assumption then implies that the difference in negligible.

We prove that game 4(i).1 is indistinguishable from game 3 under the decisional Diffie-Hellman assumption. Precisely, we aim to show that if a distinguisher $\mathcal{D}$ could efficiently distinguish between the games, then it could be used to break the DDH assumption. This implies that $\mathsf{Adv}_{4(i).1} \leq \mathsf{Adv}_3 + \max_{\mathcal{D}} \epsilon_{\mathcal{D}}$, where $\epsilon_{\mathcal{D}}$ is the probability that a PPTM $\mathcal{D}$ correctly distinguishes between Games 3 and 4(i).1 because we create a hybrid

It remains to bound $\epsilon_{\mathcal{D}}$, which we do with a reduction to decisional DH. Specifically, suppose $\mathcal{D}$ is such a distinguisher. We construct an adversary $\mathcal{A}(\mathcal{D})$ against the DDH game as follows. Given DDH challenge g^x, g^y, g^z and the challenge of determining whether or not $z = xy$, $\mathcal{A}(\mathcal{D})$ simulates the hybrid game as the challenger in a fully honest way except it inserts $g^x = g^{x_1}, g^y = g^{x_2}, g^z = g^{x_1 x_2}$. Note that g^z is the "secret" from Alice and Bob with g^{x_1} and g^{x_2} respectively, which the DDH adversary/simulator is given, so the simulator can compute all public DH intermediate keys up the tree that use g^z, including the group key at the top of the tree. This hybrid game is clearly a hybrid between Game 3 and Game $4(i).1$, with equal probability of either. The simulator answers all queries in the honest way, except in the send/create queries where it needs to insert these DH values. It knows when and where to do this because of the earlier game hops and the freshness predicate: it knows exactly where to insert them in the game and by the freshness predicate, they are honest. It is also able to answer any reveal queries because these values are ephemeral and it never has to reveal them to the adversary due to the freshness predicate. Therefore the simulation is sound.

In Game 1 we ensured no DH keys collide, and with probability $1/q$ the DDH challenger may provide challenge values $g^x = g^y$, in which case the simulator must abort. Thus we have the bound $\mathrm{adv}_3 \leq \mathrm{adv}_{4(i).1} + \epsilon_{\mathrm{DDH}} + 1/q$.

Case (i), Game $4(i).k$ **where** $2 \leq k \leq \lfloor \gamma/2 \rfloor$**.** We repeat the replacement performed in the previous game, but for the next pair of sibling nodes. Again, detecting this replacement would require violating DDH. At this point, the tree key is no longer a function of the leaf keys—instead, it depends on the keys at the nodes whose children are leaves, each of which has been replaced by a random value, unknown to the adversary.

Case (i), Game $4(i).\ell$ **where** $\ell \leq L \leq \gamma$**.** We iteratively replace DH keys using the DDH assumption, starting along the base of the tree and then working our way up until eventually all DH keys in the tree, including the final group key, are independent of each other. It is trivially impossible for the adversary to do any better than guessing in the final game. Given a group size of n, we never need to do more than $n \leq \mathsf{n_P}$ such game hops due to our tree structure. Thus

$$\mathrm{adv}_{\mathsf{n_P}} \leq \gamma\,(\epsilon_{\mathrm{DDH}} + 1/q) + 0$$

Case (ii), Game 4(ii). We now proceed with case (ii), restarting our game hopping sequence on Game 3. Assume now that E does not hold, and thus the copath in the session state of the Tested session is not fresh. Since the Tested session must be fresh, the first disjunct of the final clause of the freshness predicate must hold: that $t > 0$ and stage $(u, i, t-1)$ is fresh.

We proceed by induction on the stage number of the Test session. Our inductive hypothesis at step k is that no adversary can win stage k with non-negligible advantage. The base case $k = 0$ holds by the above argument: case (ii) cannot apply since the freshness predicate in case $k = 0$ requires E to occur.

Assume now that the inductive hypothesis is true for stage $t \leq k-1$; we show that it is also true for $t = k$. As before, if the adversary Tests (u, i, t), then this means stage t must be fresh. Let RO be the event that the adversary queried the random oracle and received the session key of the Test session as a reply.

If RO does not hold, we perform a single game hop in which we replace the session key with a random value. Since the random oracle response is by construction a random value, this replacement is indistinguishable and the resulting advantage is zero. We use the fact that the adversary is not permitted to perform any $\mathsf{RevSessKey}$ queries which return the Tested stage key.

Thus, we conclude that RO must hold; in particular, the adversary must have queried $KDF(\pi.\mathsf{sk}, \pi.\mathtt{tk}, 0)$. We are in case (ii), so the freshness predicate restricts the adversary from issuing a $\mathsf{RevSessKey}(u, i, k-1)$ query to learn the stage key of stage $k-1$. This adversary therefore has a distinguishing advantage against the previous stage, which will contradict our induction hypothesis.

Specifically, given such an adversary $\mathcal{A}$ we construct an adversary $\mathcal{A}'$ which wins with non-negligible probability against stage $k-1$. $\mathcal{A}'$ simply simulates $\mathcal{A}$ without changing any values and recording all random oracle queries; the simulation is thus trivially faithful. When $\mathcal{A}$ issues a $\mathsf{Test}(u, i, s)$ query, $\mathcal{A}'$ issues a $\mathsf{Test}(u, i, s-1)$ query and compares the resulting key to all of $\mathcal{A}$'s random oracle queries. If it appears in a random oracle query, $\mathcal{A}'$ outputs $b = 0$; otherwise, it outputs $b = 1$. By construction, stage $(u, i, s-1)$ is fresh and its session key is an argument to the random oracle, so the advantage of $\mathcal{A}'$ is non-negligible.

This contradicts our inductive hypothesis that no adversary can win against a stage less than t with non-negligible probability; the result thus holds in case (ii) by induction. □