On Ends-to-Ends Encryption:

Asynchronous Group Messaging with Strong Security Guarantees

Katriel Cohn-Gordon*[1], Cas Cremers[2], Luke Garratt[1], Jon Millican[3], and Kevin Milner[1]

[1]Department of Computer Science, University of Oxford
[2]CISPA Helmholtz Center for Information Security, Saarland Informatics Campus, Germany
[3]Facebook

Version 2.3, March 2nd, 2020†

Abstract

In the past few years secure messaging has become mainstream, with over a billion active users of end-to-end encryption protocols such as Signal. The Signal Protocol provides a strong property called post-compromise security to its users. However, it turns out that many of its implementations provide, without notification, a weaker property for *group* messaging: an adversary who compromises a single group member can read and inject messages indefinitely.

We show for the first time that post-compromise security can be achieved in realistic, asynchronous group messaging systems. We present a design called Asynchronous Ratcheting Trees (ART), which uses tree-based Diffie-Hellman key exchange to allow a group of users to derive a shared symmetric key even if no two are ever online at the same time. ART scales to groups containing thousands of members, while still providing provable security guarantees. It has seen significant interest from industry, and forms the basis for two draft IETF RFCs and a chartered working group. Our results show that strong security guarantees for group messaging are practically achievable in a modern setting.

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

†An extended abstract of this paper appears at ACM CCS 2018 [15]; this is the full version. A summary of changes is given in Appendix F.

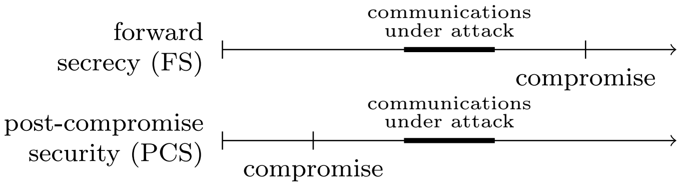

forward secrecy (FS)
communications under attack
compromise
post-compromise security (PCS)
communications under attack
compromise

Figure 1: Attack scenarios of forward secrecy and PCS, with the communications under attack marked in **bold** and time from left to right. Forward secrecy protects against *later* compromise; PCS protects against *earlier* compromise.

1 Introduction

The security of secure messaging systems has improved substantially over recent years; WhatsApp now provides end-to-end encryption for its billion active users, based on Open Whisper Systems' Signal Protocol [38, 55], and The Guardian publishes Signal contact details for its investigative journalism teams [53].

The Signal Protocol and its variants offer a security property called Post-Compromise Security (PCS) [14], sometimes referred to as "future secrecy" or "self-healing". For PCS, even if Alice's device is entirely compromised by an adversary, she will automatically re-establish secure communications with others after a single unintercepted exchange, even if she was not aware of the compromise. Thus, PCS limits the scope of a compromise, forcing an adversary to act as a permanent active man-in-the-middle if they wish to exploit knowledge of a long-term key. This can serve as a powerful impediment to mass-surveillance techniques. Thus far, PCS-style properties have only been proven for point-to-point protocols [13], and they are only achievable by stateful ones [14]. Figure 1 illustrates the difference between forward secrecy and PCS. Because it raises the bar for mass-surveillance, we see PCS as an important property for any modern secure messaging protocol.

Systems like WhatsApp and Signal are designed to be usable by anyone, not just experts, and to provide much of the same functionality as existing insecure messaging applications. To that end, they must work within a number of constraints, an important one of which is *asynchronicity*: Alice must be able to send messages 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. Asynchronicity means that standard techniques for forward secrecy, such as a DH key exchange, do not apply directly. This has driven the development of novel techniques to achieve forward secrecy without interaction, e.g., using sets of precomputed DH "prekeys" [37] that Bob uploads to a server, or by using puncturable encryption [25].

Group and multi-device messaging is important for many users, and various implementers have designed their own protocols to support it. However, since group conversations must also be asynchronous, it is not straightforward to adapt existing group key exchange (GKE) protocols, which usually require a number of interactive rounds of communication, to this context. An alternative is to use a two-party protocol between every pair of group members, but 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. In many real-world scenarios, this inefficiency is a problem, especially where bandwidth is restricted or expensive e.g., 2G networks in the developing world[1].

[1] The 2015 State of Connectivity report by internet.org [26] lists affordability of mobile data as one of

In fact, modern messaging protocols which provide PCS for two-party communications generally drop this guarantee for their group messaging implementations without notifying the users. For example, WhatsApp, Facebook Messenger and Google Allo have mechanisms to achieve PCS 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 PCS [20, 55]. Indeed, in these systems, an adversary who fully compromises a single group member can indefinitely and passively read future communications in that group (though certain events, e.g. removing a device, may cause group changes and generation of new keys). 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 group messaging solution that (i) allows participants to communicate *asynchronously*, (ii) *scales* sublinearly in the group size, and (iii) admits *strong security guarantees* such as PCS? In this paper we address this open question, and show how to devise a protocol that achieves it. Our main contributions are:

(1) We design a fully-asynchronous tree-based GKE protocol that offers modern strong security properties, called Asynchronous Ratcheting Trees (ART). ART derives a group key for a set of agents without any pair needing to be online at the same time, a requirement for modern messaging protocols. Notably, ART's properties include PCS: messages can be secret even after total compromise of an agent.

ART has seen significant interest from industry and is the basis of the IETF MLS working group and two draft RFCs [42].

(2) We give a game-based computational security model for our protocol, building on multi-stage models to capture the key updates. This allows us to encode strong properties such as PCS. We give a game-hopping computational proof of the unauthenticated core of our ART protocol, with an explicit reduction to the PRF-ODH problem, and a mechanised symbolic verification of its authentication property using the TAMARIN prover. Our hybrid argument follows e.g. [33].

(3) We present and evaluate a proof-of-concept Java implementation of ART's core algorithms, increasing confidence in the practicality 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 should allow many insights from the existing literature in (synchronous) group protocols to be applied in the asynchronous setting. We give examples, including dynamic groups, in Section 8. *We provide the proof-of-concept implementation and evaluation data at [41].*

2 Background and Related Work

There has been research into group messaging protocols for decades, and we do not aim to survey the entire field of literature. We discuss here several previous classes of approach. A key point which distinguishes our work from past research is our focus on asynchronicity and PCS; ART can derive a group key with PCS even if no two participants are ever online at the same time.

the four major barriers to global connectivity, with a developing-world average monthly data use of just 255 MB/device.

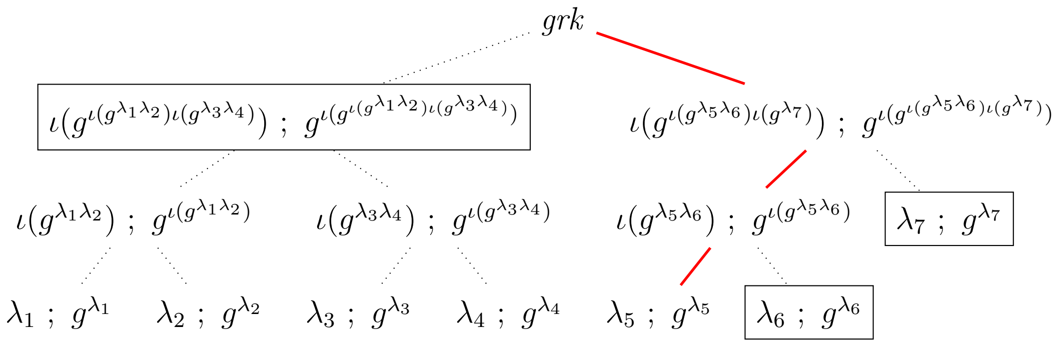

grk
$\iota(g^{\iota(g^{\lambda_1\lambda_2})\iota(g^{\lambda_3\lambda_4})})$; $g^{\iota(g^{\iota(g^{\lambda_1\lambda_2})\iota(g^{\lambda_3\lambda_4})})}$
$\iota(g^{\iota(g^{\lambda_5\lambda_6})\iota(g^{\lambda_7})})$; $g^{\iota(g^{\iota(g^{\lambda_5\lambda_6})\iota(g^{\lambda_7})})}$
$\iota(g^{\lambda_1\lambda_2})$; $g^{\iota(g^{\lambda_1\lambda_2})}$
$\iota(g^{\lambda_3\lambda_4})$; $g^{\iota(g^{\lambda_3\lambda_4})}$
$\iota(g^{\lambda_5\lambda_6})$; $g^{\iota(g^{\lambda_5\lambda_6})}$
λ_7 ; g^{λ_7}
λ_1 ; g^{λ_1}
λ_2 ; g^{λ_2}
λ_3 ; g^{λ_3}
λ_4 ; g^{λ_4}
λ_5 ; g^{λ_5}
λ_6 ; g^{λ_6}

Figure 2: Example DH tree. We mark each node with its secret and public keys, separated by ; . Recall that the λ_j denote leaf keys, and $\iota(\cdot)$ denotes the mapping from group elements to integers. The path from λ_5 to the root is marked in solid red, and the $\boxed{\text{boxed}}$ nodes lie on its copath. An agent who knows λ_5 and the public keys on its copath can compute *grk* by repeated DH operations.

2.1 Other Group Messaging Protocols

OTR-style Goldberg, Ustaoglu, Van Gundy, and Chen [23] define Multi-Party Off the Record Messaging (mpOTR) as a generalisation of the classic OTR [6] protocol, aiming for security and deniability in online messaging. mpOTR has since given rise to a number of interactive protocols such as $(N+1)$sec [19]. The general design of this family 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.

Assuming an authentic network [11, 52] discuss "asynchronous" GKE in the setting of distributed systems, in the sense that they do not rely on a centralised clock. They require several interactive rounds of communication, and do not provide PCS.

Physical approaches Some work uses physical constraints to restrict malicious group members. For example, HoPoKey [40] has its participants arrange themselves into a circle, with neighbours interacting. This allows it to derive strong security properties. We, however, will not assume physical co-location.

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 [55]. 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). This variant does not have PCS.

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

***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 $g^{x_0}, \ldots, g^{x_n}$ and a single x_i ($i \leq n$), derive a value *grk* which is hard to compute without knowing one of the x_i. With

$n = 2$ this can be achieved by traditional DH, and with $n = 3$ Joux [27] gives a pairing-based construction. However, for general n construction of such a primitive is a known open problem. [4] essentially generalise the Joux protocol with a construction from an $(n - 1)$-non-degenerate linear map on the integers, and [5, 36] construct it from iO.

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, Strawczynski, Diffie, and Wiener [51], and the seminal academic work is perhaps Wallner, Harder, and Agee [54] or Wong, Gouda, and Lam [56]. Later examples include [7, 9, 12, 18, 28, 29, 30, 31, 32, 35, 44, 45, 57], among many others. These protocols assign private DH keys to leaves of a binary[2] tree, defining (i) g^{xy} as the secret key of a node whose two children have secret keys x and y, and (ii) $g^{g^{xy}}$ as its public or 'blinded' key. 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. We depict a generic DH tree in Figure 2.

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 the mapping from group elements to integers.

The online exchanges in these protocols are due to, at least in part, the requirement for agents to know the public keys on their copath. For example, in Figure 2 on the preceding page, node 5 must know (but cannot compute just from the g^{λ_j}) all $\boxed{\text{boxed}}$ public keys. Other agents may be chosen by the messaging system to compute and broadcast public keys at intermediate nodes; for example, Kim, Perrig, and Tsudik [32] describe a system of subtree "sponsors" who broadcast select public keys. However, none of these solutions provide PCS, because they do not support updating keys.

2.2 Deployed Implementations

Several widely-used mobile apps deploy encrypted group messaging protocols. We survey some of the most popular, giving asymptotic efficiencies for three main designs in Table 1 on page 7. In concurrent work, [46] examine the group messaging protocols used by WhatsApp, Signal and Threema, finding a number of vulnerabilities related to their group operations.

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; 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 Alice cannot use WhatsApp from any device if her phone is offline.

[2] Some constructions use ternary trees; the underlying concept is the same.

Facebook Messenger Secret Conversations similarly uses the Sender Keys variant of Signal for all conversations involving 3+ devices [20]. As in the WhatsApp implementation, Sender Keys are only rotated when a device is removed from a conversation.

Apple iMessage uses 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 practical on a 2009 iPhone 3GS.

Signal The Signal mobile application uses pairwise Signal channels for group messaging[3], with additional devices on a Signal account implemented as first class participants.

SafeSlinger [21] is a secure messaging app whose goal is usable, "privacy-preserving and secure group credential exchange". It aims for message secrecy under an adversary model that allows for malicious participants. The two greatest differences between ART and SafeSlinger are *security goals* and *synchronicity*. First, ART is explicitly designed to achieve PCS of message keys, while SafeSlinger instead aims for (non-forward) secrecy and just derives a single group key. Of necessity [14], ART must therefore support stateful and iterated key derivations. Using SafeSlinger's unbalanced DH key tree with ART's key updates, while reducing the computational load on the initiator, would take linear (versus logarithmic) time. Second, SafeSlinger is a synchronous protocol with commitment, verification and secret-sharing rounds, in which all group members must be online concurrently. ART, on the other hand, is an asynchronous protocol which supports messaging offline members.

3 Objectives

Security properties for AKE protocols are extremely well-studied. We now describe our high-level threat model and security goals.

Secrecy and Authentication Our fundamental goal is confidentiality and authenticity of keys: an active adversary should not learn keys shared between Alice and Bob.

Post-Compromise Security (PCS) 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, Cremers, and Garratt [14] defined PCS 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: if the full state of a group member is compromised (long-term and other derived keys) but the group conversation continues without interference, the resulting group key should 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. PCS is a major distinguishing feature of modern two-party messaging protocols, and offers significant protection from adversaries with large resources, forcing them to actively interfere in all communications even after they manage to temporarily compromise a device.

Poor randomness Security models such as extended Canetti-Krawczyk and its generalisations [16, 34] allow revealing random numbers generated by a party whose long-term keys

[3] Based on source code inspection [43].

Table 1: Asymptotic efficiencies and properties of some group messaging solutions as a function of the group size n. "Pairwise Signal" denotes sending a group message repeatedly over individual Signal channels. In the setup phase, the values here refer to the total work done for all users to reach a point where no further setup is required and "sender" refers to the creator of the group. In the sending phase, "other" refers to a recipient of a message. We provide concrete measurements in Section 7.

		number of exponentiations		number of symmetric encryptions		bandwidth		PCS
		sender	per other	sender	per other	sender	per other	
sender keys	setup	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	✗
	ongoing	0	0	1	1	$O(1)$	$O(1)$	
pairwise Signal	setup	$O(n)$	$O(n)$	0	0	$O(n)$	$O(n)$	✓
	ongoing	$O(n)$	$O(1)$	$n-1$	1	$O(n)$	$O(1)$	
our solution	setup	$O(n)$	$O(\log(n))$	0	0	$O(n)$	$O(n)$	✓
	ongoing	$O(\log(n))$	$O(\log(n))$	1	1	$O(\log(n))$	$O(\log(n))$	

are uncompromised. However, many widely-used protocols (such as TLS 1.3 or Signal) do not achieve this. Here we aim for some security in the face of revealed randomness.

3.1 Security properties

Informally, we want our messaging protocol to provide *implicit authentication* and *message secrecy* against various strong adversaries:

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

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

PCS [14]. 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.

PCS is a goal which previous work does not aim for. As discussed earlier, without PCS an adversary who compromises one participant may be able to intercept group messages indefinitely, a property which does not hold of two-party Signal communications.

3.2 Properties Out of Scope

Our goal in this work is to provide a provably-secure design for an asynchronous group messaging system. In the interest of transparency, therefore, we discuss here some important messaging-related problems which we do not set out to solve, referring the reader to other research or designs. By "out of scope" we do not mean that these problems are unimportant or their solutions unnecessary—rather, merely that we are not setting out to solve them in this work. In many cases, a solution will indeed be necessary in a large-scale practical deployment. As we will see later, our designs build on well-studied DH tree based systems, thereby enabling the reuse of existing solutions as components.

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 send, 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 sender keys' explicit signatures. We choose the simpler option and do not include signature keys, discussing this topic 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. 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.

3.2.2 Malicious group members

In the two-party case, traditional security properties generally assume that the peer to a session is honest. With n parties, there is an intermediate condition: when $m < n$ members of the group are honest. For example, Abdalla, Chevalier, Manulis, and Pointcheval [1] give a GKE 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 design to handle group membership changes should be relatively straightforward. We discuss dynamic groups further in Section 8.

Trust in the Initiator A particular example of a malicious insider is the group creator, who may be able to choose malicious initial values. For example, a malicious creator might be able to secretly add an eavesdropper to a group without revealing their presence to the other (honest) group members. (Note that they could of course just publish the received messages, regardless of the protocol.) As for any other group member, we consider this attack out of scope.

ART's asynchronicity constraint means that Alice must be able to send a message to a group she has just created, even if none of the other participants have yet been online. ART's design allows for this, but at a cost: if Alice is corrupted during this initial phase, the resulting stage keys are insecure until all group members have performed an update. We capture this increased requirement in our freshness predicates, and note that one can remove it if all participants are online, by having each one in turn perform a key update. Our approach here is related to that of the zero round-trip (0RTT) mode of TLS 1.3, in which agents can achieve asynchronicity at the cost of a weaker security property for early messages.

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. In particular, if Alice and Bob both send a key update at the same time, only one can consistently be applied; this does not violate any secrecy properties, but may break availability if updating a key is necessary to send a message. We remark on two main techniques to avoid trivial denials of service, though a perfect solution is an open research question (studied e.g. by [12]) and we consider it out of scope for our work.

The first technique is to decouple state updates from message sending: once Bob has derived a valid sending key, the protocol may accept messages sent under that key for a short duration even if Bob should have performed a state update. This allows Bob to send messages immediately while in the background performing a recovery process to return to the latest group state, at the cost of weakened security guarantees due to the extended key lifetime.

A second solution is at the transport layer, either by enforcing in-order message delivery or by refusing to accept out-of-order key updates and instead delivering the latest group state. That is, when the transport layer server receives a state update from Bob which was generated based on an out-of-date state, it can refuse to accept it and instead instruct Bob to process the latest updates and retry. Since this enforcement can operate based only on message metadata, a malicious transport server can then violate availability but not message confidentiality or integrity. This solution works fine for many group sizes, but in very large groups may cause a server performance bottleneck.

3.2.4 Transcript agreement

In many scenarios it is valuable for all group participants to 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 cover it here.

4 Notation

We write $x := y$ to denote assigning y to the variable x. We write $x :=_\$ S$ to denote sampling a random element from the distribution S and assigning it to the variable x; in particular, S may be the output distribution of a randomised function f.

DH groups We work in a DH group $\mathcal{G}$ (with generator g) admitting a mapping $\iota(\cdot) : \mathcal{G} \to \mathbb{Z}/_{|G|\mathbb{Z}}$ from group elements to integers, allowing us to interpret a group element g^x itself as the secret key corresponding to a new public key $g^{\iota(g^x)}$. 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; thus for example the public setup key SUK is defined to be g^{suk}. We denote by DHKeyGen a randomised algorithm returning a private key in the DH group. To separate the ART initial key exchange from the subsequent tree operations, we define a distinct key generation algorithm KeyExchangeKeyGen that also returns a private DH key.

We assume that the following $\mathsf{PRF\text{-}ODH}$ problem is hard: given a tuple (g^x, g^y, z_b) where $z_0 := \iota(g^{xy})$ and $z_1 := \iota(g^z)$ for uniformly randomly chosen z, the advantage of any PPT distinguisher in outputting b is negligible.

Signatures and MACs ART uses two explicit authenticators: a signature to authenticate the initial group setup message, and a MAC to authenticate subsequent updates. $s = \textsc{Sign}(m, sk)$ denotes a signature of the message m with the private key sk, and $\textsc{SigVerify}(m, s, pk)$ verifies the signature against a public key pk, returning a boolean representing whether the verification succeeds. $\mu = \text{MAC}(m, k)$ is a MAC of the message m with the symmetric key k, and $\text{MAC}\textsc{Verify}(m, \mu, k)$ verifies it and returns a boolean.

Trees We define binary trees as a combination of nodes (which contain two nested children) and leaves (which contain no children), along with associated data at each node and leaf: $\mathrm{tree} ::= (\mathrm{node}(\mathrm{tree}, \mathrm{tree}), \cdot) \mid (\mathrm{leaf}, \cdot)$. For a binary tree T, we use the notation $|T|$ to refer to the total number of leaves in the tree. We label each node of a tree with an index pair (x, y), where x represents the level of the node: the number of nodes (excluding the node 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 $(x + 1, 2y + 1)$. We write $T_{x,y}$ for the data at index (x, y) in a tree T. All tree nodes but the root have a parent node and a sibling (the other node directly contained in the parent). We refer to the *copath* of an node 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 2. Finally, we usually associate a secret key x and corresponding public key g^x to each node, which we denote by labelling nodes with $x \ ; \ g^x$.

Derived Keys ART contains various types of key:

Leaf keys λ_j are secret DH keys assigned to tree leaves.

Node keys nk are secret DH keys assigned to non-leaf 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.

Note that stage keys sk play the role of "root keys" in the two-party Signal protocol. We avoid the term "root" to prevent confusion with the root of the DH tree.

5 Design

We now present ART's core designs, in two parts.

First, we give a tree-based group DH protocol related to those from Section 2.1, but unlike those protocols ours is the first *fully asynchronous* design. In other words, it is possible for all group members including the creator to conduct the key exchange protocol and derive the shared group key without waiting for any other group member to respond to a message. This is necessary in order to use a group AKE protocol in practical deployments, where group members may be offline due to e.g., unreliable mobile network connections.

Second, we define an efficient protocol for an ART group member to *update* their personal keys and establish a new shared group key, using the underlying tree structure in the group. This update protocol enables Post-Compromise Security: if a group member's local state is compromised but they are later able to perform an update without adversarial interference, then the group key derived after their update will once again be secret and authentic. ART is the first group AKE protocol to provide PCS.

Informal explanations of our algorithms follow in Sections 5.1 and 5.2, and formal definitions in pseudocode are presented in Appendix A. Example trees are shown in Figure 3.

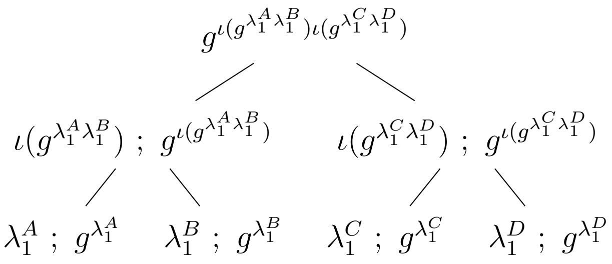
5.1 ART Construction

The main reason that we cannot directly deploy a tree DH protocol is that the initiator may be the only online member when creating a group. Indeed, in a four-person group, even if Alice has public keys to use for the three other leaves, she cannot compute the public key of the parent node of C and D (marked below as "**?**").

Alice generates a new ART tree with

$$\lambda_1^B = \textsc{KeyExchange}(ik_a, IK_b, suk, EK_b)$$
$$\lambda_1^C = \textsc{KeyExchange}(ik_a, IK_c, suk, EK_c)$$
$$\lambda_1^D = \textsc{KeyExchange}(ik_a, IK_d, suk, EK_d)$$

and broadcasts all the public keys. KeyExchange must be a strong one-round AKE protocol but ART does not have specific requirements on its structure.



$g^{\iota(g^{\lambda_1^A\lambda_1^B})\iota(g^{\lambda_1^C\lambda_1^D})}$
$\iota(g^{\lambda_1^A\lambda_1^B})$; $g^{\iota(g^{\lambda_1^A\lambda_1^B})}$
$\iota(g^{\lambda_1^C\lambda_1^D})$; $g^{\iota(g^{\lambda_1^C\lambda_1^D})}$
λ_1^A ; $g^{\lambda_1^A}$
λ_1^B ; $g^{\lambda_1^B}$
λ_1^C ; $g^{\lambda_1^C}$
λ_1^D ; $g^{\lambda_1^D}$

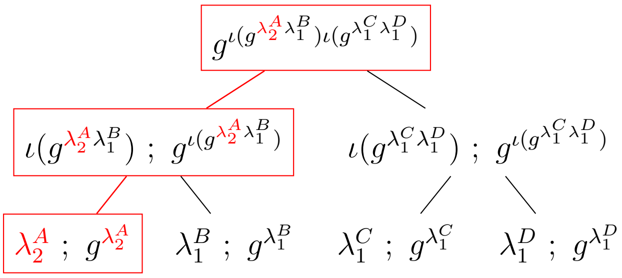
(a) Alice creates an ART group with three other members.

Alice updates their key by choosing a new leaf key λ_2^A, computing the updated nodes

$$g^{\lambda_2^A\lambda_1^B} = \left(g^{\lambda_1^B}\right)^{\lambda_2^A}$$

$$g^{\iota(g^{\lambda_2^A\lambda_1^B})\iota(g^{\lambda_1^C\lambda_1^D})} = \left(g^{\iota(g^{\lambda_1^C\lambda_1^D})}\right)^{\iota(g^{\lambda_2^A\lambda_1^B})}$$

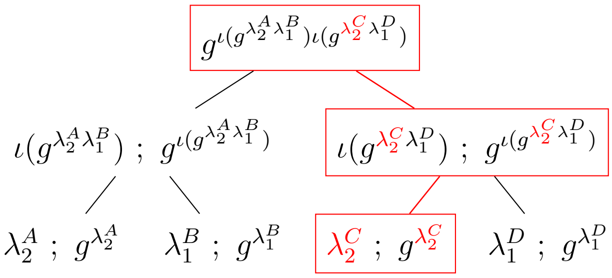
on the path from λ_2^A to the tree root, and broadcasting the updated public keys to the group.



$\boxed{g^{\iota(g^{\lambda_2^A\lambda_1^B})\iota(g^{\lambda_1^C\lambda_1^D})}}$
$\boxed{\iota(g^{\lambda_2^A\lambda_1^B}) ; g^{\iota(g^{\lambda_2^A\lambda_1^B})}}$
$\iota(g^{\lambda_1^C\lambda_1^D})$; $g^{\iota(g^{\lambda_1^C\lambda_1^D})}$
$\boxed{\lambda_2^A ; g^{\lambda_2^A}}$
λ_1^B ; $g^{\lambda_1^B}$
λ_1^C ; $g^{\lambda_1^C}$
λ_1^D ; $g^{\lambda_1^D}$

(b) Alice updates, choosing a fresh leaf key and broadcasting updated public keys. Updated nodes are shown in $\boxed{\text{boxed red}}$

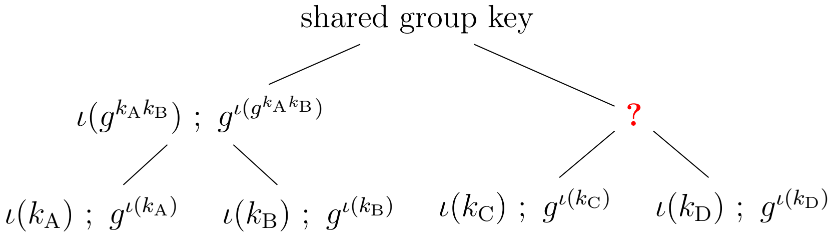
Charlie updates their key in the same way: by choosing a new leaf key λ_2^C, computing the updated nodes on the path from λ_2^C to the tree root, and broadcasting the updated public keys to the group.



$\boxed{g^{\iota(g^{\lambda_2^A\lambda_1^B})\iota(g^{\lambda_2^C\lambda_1^D})}}$
$\iota(g^{\lambda_2^A\lambda_1^B})$; $g^{\iota(g^{\lambda_2^A\lambda_1^B})}$
$\boxed{\iota(g^{\lambda_2^C\lambda_1^D}) ; g^{\iota(g^{\lambda_2^C\lambda_1^D})}}$
λ_2^A ; $g^{\lambda_2^A}$
λ_1^B ; $g^{\lambda_1^B}$
$\boxed{\lambda_2^C ; g^{\lambda_2^C}}$
λ_1^D ; $g^{\lambda_1^D}$

(c) Charlie updates their key in the same way.

Figure 3: Example ART tree creation and updates. We write secret keys and the corresponding public keys at each node except the root, separated by $\|$. Leaf keys are denoted λ_i^u, where u is the corresponding identity and i a counter. $\iota(\cdot)$ denotes a mapping from group elements to integers. From any secret leaf key and the set of public keys on its copath, an agent can compute the tree key by repeated exponentiation.


shared group key
$\iota(g^{k_\mathrm{A} k_\mathrm{B}})$; $g^{\iota(g^{k_\mathrm{A} k_\mathrm{B}})}$
?
$\iota(k_\mathrm{A})$; $g^{\iota(k_\mathrm{A})}$
$\iota(k_\mathrm{B})$; $g^{\iota(k_\mathrm{B})}$
$\iota(k_\mathrm{C})$; $g^{\iota(k_\mathrm{C})}$
$\iota(k_\mathrm{D})$; $g^{\iota(k_\mathrm{D})}$

Our insight here is that the initiator should not directly use received public keys at the leaf nodes, since then they cannot derive their parents' public keys. Instead, we propose a design in which they derive secret keys for each leaf node with the properties that

(i) for each group member, both the initiator and that group member can asynchronously derive the secret key assigned to that leaf node but

(ii) no other actor (group member or adversary) can derive that secret key.

The creator can use their knowledge of all the secret keys to compute the intermediate public keys in the tree, subsequently deleting the leaf secrets. (This leads to additional trust assumptions on the initiator, but assumptions which are mitigated by the key update protocol we define later.)

How can we derive these leaf secrets? Our core insight is that they can be the session keys of any strong one-round AKE protocol, which we denote KeyExchange. KeyExchange takes two private keys *ek* and *ik* and two public keys *EK* and *IK* and returns a bitstring, with the property that $\textsc{KeyExchange}(ik_I, IK_R, ek_I, EK_R) = \textsc{KeyExchange}(ik_R, IK_I, ek_R, EK_I)$.[4]

To use such a protocol we leverage the existing idea of *prekeys* and introduce the new idea of a *setup key*. Prekeys were first introduced by Marlinspike [37] for asynchronicity in the TextSecure messaging app. They are DH ephemeral public keys 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, and act as initial messages for a one-round AKE protocol.

To enable computing an initial tree for a group, we introduce the idea of a one-time DH *setup key*, generated locally by the creator of a group and used only during the creation of that session. 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. Suppose the initiator ("Alice") wishes to create a group of size n containing herself and $n-1$ peers. She begins by generating a DH key *suk* we call the setup key. She then requests from the public-key infrastructure the public identity key *IK* and an ephemeral prekey *EK* for each of her intended peers ("Bob", "Charlie", ...), numbering them 1 through $n-1$. Using her secret identity key ik_a and the setup key *suk* together with the received public keys for each peer, she executes a one-round key exchange protocol to derive leaf keys $\lambda_1, \ldots, \lambda_{n-1}$. Using these generated leaf keys together with a fresh leaf key λ_0, she builds a DH tree whose root becomes the initial group key.

We do not force a particular instantiation of KeyExchange. For example, it can be instantiated 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 (discussed in Section 6.3) is with a strong AKE protocol which provides authentication, in which case the group key can inherit the authentication properties.

To create a group, Alice broadcasts

[4] If the AKE protocol has a different algorithm for the initiator and responder, we can add an additonal 'Role' argument.

(i) the public prekeys (EK_i) and identities (IK_i) she used,
(ii) the public setup key SUK,
(iii) the tree T of public keys, and
(iv) a signature of (i),(ii),(iii) under her identity key.

Upon receiving such a message and verifying the signature, each group member can reproduce the computation of the tree key. First, they compute their leaf key $\lambda_i = \textsc{KeyExchange}(ik_i, IK_A, ek_i, SUK)$. Second, they extract their copath of public keys from the tree. Finally, they iteratively exponentiate with the public keys on the copath until they reach the final key, which by construction is the shared secret at the root of the tree. (Recall that we call this shared secret the "tree key" tk, and derive from it the stage key sk.)

We give a pseudocode definition of these algorithms in Figure 8 on page 29, Algorithms 1,2 and 3.

5.2 ART Updates

To achieve PCS, we must be able to *update* stage keys in a way that depends both on state from previous stages and on newly exchanged messages. (Cohn-Gordon, Cremers, and Garratt [14] prove necessity of this double dependency.) Since PCS is an explicit goal of ART, it must therefore support an efficient mechanism for any group member to update their key.

If e.g., Alice changes her leaf key, other group members can compute all the intermediate values in the resulting updated tree using only (i) their view of the tree before the change, and (ii) the list of updated public DH keys of nodes along the path from Alice's leaf node to the root of the tree. This update is efficient and asynchronous, since Alice can compute (ii) in logarithmic time and broadcast it to the group with her new leaf key.

Specifically, if at any point Alice wishes to change her leaf key from λ_b to λ'_b, she computes the new public keys at all nodes along the path from her leaf to the tree root, and broadcasts to the group her public leaf key together with these public keys. She authenticates this message with a MAC under a key derived from 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, and can be done purely locally.

We give a pseudocode definition of these algorithms in Figure 8, Algorithms 4 and 5.

Stage key chaining In order to achieve PCS, stage keys cannot be independent—instead, each stage key must depend on both the recent message exchange and on previous stages. As long as one of these two sources of secret data is unknown to the adversary, the stage key will be as well. (Cohn-Gordon, Cremers, and Garratt [14] prove an impossibility result that no stateless protocol can achieve PCS, giving a generic attack.) The resulting stage keys form a hash chain as depicted in Figure 4.

5.3 Algorithms

We give pseudocode algorithms for all of the operations in our design in Figure 8 on page 29. 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 suk, and a leaf keypair with secret key λ_0 for herself. She retrieves the public identity and ephemeral prekeys of each peer, and creates the tree in Figure 5.

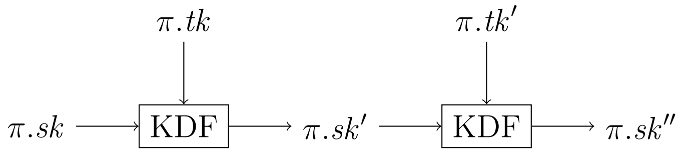


Figure 4: Derivation of stage keys $\pi.sk$. When a new tree key $\pi.tk$ is computed (as the root of a DH tree), it is combined with the current stage key to derive a new stage key $\pi.sk'$, etc. This “chaining” of keys is an important ingredient for achieving PCS. Note that the ART KDF also includes π.IDs and $\pi.T$, per Algorithm 2 on page 29.

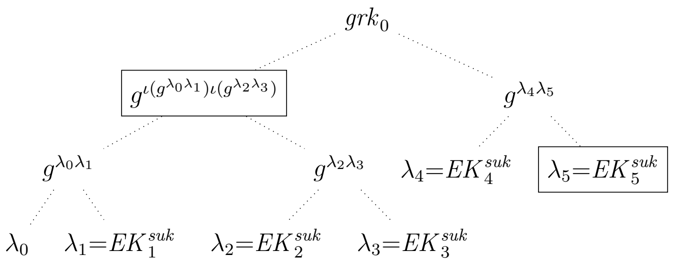


Figure 5: Alice sets up a new tree with herself and five other agents. The copath of Agent 4 is shown boxed.

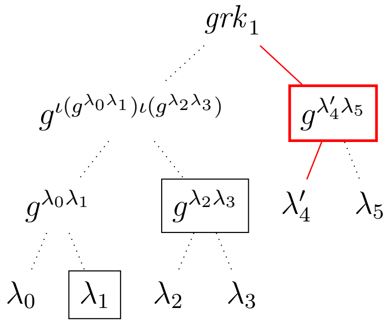


Figure 6: Agent 4 updates their leaf key. The path of Agent 4 is shown in solid red, and the copath of Alice (Agent 0) is shown boxed.

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 index 4 would receive

$$4, IK_{\text{Alice}}, IK_1, \ldots, IK_5, EK_4, SUK, g^{\iota(g^{\iota(g^{\lambda_0\lambda_1})\iota(g^{\lambda_2\lambda_3})})}, g^{\lambda_5}.$$

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 via DeriveStageKey in Algorithm 2.

Parsing this message (Algorithm 3) allows Edward to identify his position in the group tree, and to construct the group key using DeriveStageKey. 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 6. He then sends the key update message $4, g^{\lambda'_4}, g^{\iota(g^{\lambda'_4\lambda_5})}$ comprising his index as well as the path of public keys excluding the root, stores the updated leaf key and tree key, and computes the new stage key with DeriveStageKey.

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, she invokes DeriveStageKey to compute 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 GKE 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 PCS, without focusing 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 game-hopping.

Second, we show that authentication can be 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 using the X3DH protocol [39] (extended with the static-static DH key to provide more resilience against bad randomness and the KCI attack described in [33]), and verify its authentication property using the Tamarin prover, a "security protocol verification tool that supports both falsification and unbounded verification in the symbolic model" [47]. Here, we model the tree construction as a "black-box" function of the leaf keys.

Remark 1 (On the choice of model). We adopt this approach because we believe that ART's complexity is beyond the scope of current computational proof techniques. In particular, our freshness condition is already fairly complex, and its interaction with a modern AKE model with identity key corruptions leads to a state space explosion in the proof's cases.

We believe there is valuable future work to be done by increasing our model accuracy, for example by developing a systematic approach to covering the distinct cases above, or by adopting an ACCE-style definition to explicitly capture signatures and MACs.

6.1 Computational Model

We build on the multi-stage definition of Fischlin and Günther [22], in which sessions admit multiple stages with distinct keys and the adversary can Test any stage. We extend their definition 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 makes 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 and stage, receiving—uniformly at random—either its true key or a random key from the same distribution. It must then decide which it has received, winning the game if it is correct. Thus, a protocol secure in this model enjoys the property that an adversary cannot tell its true keys from random.

Similar key exchange security models generally use Activate and Run queries for the adversary to interact with the protocol algorithms. With these queries, however, there is no clear way for them to instruct agents to choose one of multiple possible actions—for example, whether or not to perform a key update. In order to clarify the distinction, we split the traditional Run algorithm into PRecv ("protocol receive", to receive and process a message from $\mathcal{A}$) and PSend ("protocol send", to receive instructions from and then send a message to $\mathcal{A}$).

Apart from this split, we use a standard set of queries and give their precise details in Table 2 on page 30. The queries comprise Create, ASend and ARecv (which allow the adversary to interact with honest participants); RevSessKey and RevRandom (which model corruption of keys used in the protocol); and Test and Guess (which are used in the security game). Since we work in an unauthenticated model, we do not need a RevLTK query.

Sessions and stages Agents may have multiple parallel conversations with various peers. We refer to a *session* as a local, long-lived communication at a particular agent; for example, Alice may have a session with peers Bob and Charlie. Sessions at an agent u are uniquely zero-indexed in creation order; thus for example we can refer uniquely to Alice's fourth session by the pair $(u, i) = (\text{Alice}, 3)$.

Sessions are updated in *stages* over time, as messages are exchanged and updates processed. Stages of a session are zero-indexed in time order, so e.g., we denote the initial stage of session $(\text{Alice}, 3)$ by the *session identifier* or *sid* $(\text{Alice}, 3, 0)$. Later stages of $(\text{Alice}, 3)$ are then denoted $(\text{Alice}, 3, 1)$, $(\text{Alice}, 3, 2)$, and so on.

Definition 1 (Session state)**.** For agent u, session counter i and stage counter t, the *session state* π comprises:

(i) $\pi.u$, the identity u of the current agent
(ii) $\pi.ik$, the identity key of the current agent
(iii) $\pi.ek$, the ephemeral prekeys of the current agent
(iv) $\pi.\lambda$, the leaf key of the current stage
(v) $\pi.tk$, the tree key of the current stage
(vi) $\pi.sk$, the stage key of current stage
(vii) $\pi.T$, the current tree (with ordered nodes) with *public* keys stored at each node
(viii) $\pi.\text{idx}$, the position of the current agent in the group
(ix) $\pi.\text{IDs}$, an ordered list of agent identifiers and leaf keys for the group, where the index of each entry is the index of the corresponding leaf in the tree
(x) $\pi.\bar{P}$, the copath of the current agent

Where considering multiple distinct session states, we refer to $\pi = \pi(u, i, t)$ as the state of the t^{th} stage of agent u's i^{th} session.

Values in π roughly correspond to variables in a protocol implementation. However, for the security definitions we also keep track of some additional "bookkeeping" state σ. Values in σ are only used for the security game, and do not correspond to variables in a protocol implementation.

Definition 2 (Bookkeeping state)**.** For agent u, session counter i and stage counter t, the bookkeeping state σ of (u, i, t) is an ordered collection of the following variables.

(i) $\sigma.i$, the index of the current session among all sessions with the same agent

(ii) $\sigma.t$, the index of the current stage in the session (initialised to 0 and incremented after each new stage key is computed)

(iii) σ.status, the execution status for the current stage. Takes the value `active` at the start of a stage, and later set to either `accept` or `reject` when the stage key is computed

(iv) σ.HonestKeys, the set of ephemeral keys honestly generated in the current stage

(v) $\sigma.\ell[i']$, the number of leaf keys received so far from node i' in $\pi.T$ (when $i' = \pi.\text{idx}$, this is the number of leaf keys that (u, i) has generated so far).

Definition 3 (sid)**.** By $\mathit{sid}(\pi, \sigma)$ we mean the triple $(\pi.u, \sigma.i, \sigma.t)$. Agents are unique, session counters monotonically increase and session state does not change without the stage changing. Therefore, such a tuple (u, i, t) uniquely identifies states π and σ if they exist.

Definition 4 (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) $(x, g^x) :=_{\$} \text{KeyExchangeKeyGen}()$: generate DH keys

(ii) $\text{Activate}(x, \rho, \mathit{peers}) \to (\pi, \sigma)$: the challenger initialises the protocol state of an agent u by accepting a long-term secret key x, a role ρ and a list peers of peers, creating states π and σ, assigning $\sigma.i$ to the smallest integer not yet used by u, and returning (π, σ)

(iii) $\text{PRecv}(\pi, m) \to \pi'$: an agent receives a message m, updating their protocol state from π to π'

(iv) $\text{PSend}(\pi, d) \to \pi', m$: an agent receives some instructions d and sends a message m, updating their protocol state from π to π'

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

6.2 Analysis: Unauthenticated Protocol

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 key from any stage, but rule out trivial attacks created by such corruptions. We define

$$\textsc{KeyExchange}(\pi.ik, \text{IDs}_0, ek, SUK) := SUK^{ek}.$$

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.

We define PSend(π, d) as follows. First, validate that d is one of "create-group" or "update-key", or else abort, setting the session state to `reject`. Then, if d is "create-group", execute the initiator's setup algorithm from Section 5.1; if d is "update-key", execute the initiator's update algorithm from Section 5.2. These algorithms are given formally as SetupGroup and UpdateKey in Section 8 on page 29.

We define PRecv(π, m) as follows. For a session with $\sigma.t = 0$, validate that m is of the expected format, and if so then extract from it the relevant tree data and execute the responder's setup algorithm defined in Section 5.1. For a session with $\sigma.t > 0$, again validate m but execute the responder's update algorithm defined in Section 5.2. These algorithms are given formally as ProcessSetupMessage and ProcessUpdateMessage in Section 8 on page 29.

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

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 $\Lambda = \lambda_0 \ldots \lambda_{n-1}$ be a list of group elements representing leaf keys. We say that $\bar{P}$ is the i^{th} *copath induced by* Λ precisely if, in the DH tree induced by Λ, each $\bar{P}_j$ is the sibling of a node on the path from λ_i to the tree root, and that $\bar{P}$ is *induced by* Λ if for some i it is the i^{th} copath induced by Λ.

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

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

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

   (a) there exists a stage with $sid(\pi, \sigma) = (u, i, t)$ such that $(\lambda_j, sid(\pi, \sigma)) \in \sigma$.HonestKeys, 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 Λ.

Definition 7 (Freshness of a stage)**.** We say that a stage with sid (u, i, t) deriving key sk is *fresh* if

(i) it has status `accept`,

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

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

(iv) one of the following criteria holds:

   (a) the current copath is fresh, or

   (b) $t > 0$ and the stage with sid $(u, i, t - 1)$ is fresh.

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

Remark 2 (Freshness of the group creator's first stage). Our freshness predicate encodes stronger trust assumptions on the initiator's first stage than it does on subsequent updates, as discussed in Section 3.2.2. Indeed, by criterion 7(iv-b) the creator's first stage is fresh only if their first copath is fresh. This copath is induced by the initial λ_j, which are added to σ.HonestKeys during the creator's first stage. Thus, by criterion 6(ii-a), if the adversary issues a RevRandom query against that stage then it will no longer be fresh. This is true until all the λ_j from the initial stage have been replaced, at which point criterion 6(ii) is fulfilled by the stages replacing them.

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

Authentication states that if the ephemeral keys used in a stage are from an uncorrupted stage 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.

Forward secrecy is captured through clause (iv)a and the definition of the $\mathsf{RevRandom}$ query. Indeed, suppose Alice accepts a stage t and then updates her key in stage $t+1$. An adversary who queries $\mathsf{RevRandom}(\ldots, t+1)$ 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)b. Indeed, suppose the adversary has issued $\mathsf{RevRandom}$ queries against all of one of Alice's session'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)b stages $j+1, j+2, \ldots, t$ are fresh as well. Our model thus requires their keys to be indistinguishable from random.

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 key of the Tested stage is indistinguishable from random. Thus, after the $\mathsf{Test}(u, i, t)$ query, the challenger flips a coin $b :=_{\$} \mathrm{Uniform}(\{0, 1\})$. With probability $1/2$ (when $b = 0$) it reveals the actual stage key of user u's ith session at stage t to the adversary, and with probability $1/2$ (when $b = 1$) it 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 the security experiment 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 a proof.

Remark 3 (Partnering experiment). Our freshness condition is separated into two parts, indistinguishability and partnering security, following the style of Brzuska, Fischlin, Warinschi, and Williams [10]. In this setting, indistinguishability is proved under the restriction that $\mathsf{RevSessKey}$ queries cannot output the Tested session key, and a separate game is used to show a form of authentication: that the session key is only derived by sessions which "should" derive it. In our context, because (i) we are working in an unauthenticated model, and (ii) all of the values upon which participants should agree are included as arguments to the session key KDF, in this case the partnering experiment and its corresponding security bound mostly consist of administrative bookkeeping. They appear in Appendix C.

Theorem 1. *Let* $\mathsf{n_P}$, $\mathsf{n_S}$ *and* $\mathsf{n_T}$ *denote bounds on the number of parties, sessions and stages in the security experiment respectively. Under the* $\mathsf{PRF\text{-}ODH}$ *assumption with KDFs modeled as* *random oracles, the success probability of any PPT adversary against the security experiment for our protocol is bounded above by*

$$\frac{1}{2} + \frac{\binom{\mathsf{n_P n_S n_T}}{2}}{q} + \gamma(\mathsf{n_P n_S n_T}^2)^\gamma \left(\epsilon_{\mathsf{PRF\text{-}ODH}} + {}^1\!/_q\right) + \mathsf{negl}(\lambda)$$

where $\epsilon_{\mathsf{PRF\text{-}ODH}}$ bounds the advantage of a PPT adversary against the PRF-ODH *game. (This bound depends only on $\epsilon_{\mathsf{PRF\text{-}ODH}}$ and not* KeyExchange *because it is unauthenticated.)*

Proof sketch (full proof in Appendix C). 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 related to one another, we are able to bound the original success probability of the adversary.

We make one modification to the protocol for technical reasons: as specified, ART has agents authenticate group creation messages with a signature under the identity key, and update messages with a MAC on a key derived from the stage key. Because these keys are also used in the key exchange protocol, we cannot achieve key indistinguishability notions of security. In the computational proof, we will therefore drop the explicit authenticators from the protocol and enforce authentication through the freshness condition instead.

The overall structure of the proof is as follows. First, we perform some administrative game hops to avoid DH key collisions. Then, we guess the indices (u, i, t) of the sid of the Test session and stage. If it is not fresh then the adversary loses. If it is fresh, we perform a case distinction based on which clause of the freshness predicate it satisfies: either the current copath 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 $\iota(g^{xy})$, and thus by hardness of the PRF-ODH problem we can indistinguishably replace it with $\iota()$ of a uniformly randomly chosen 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 PRF-ODH advantage. After all non-leaves have been replaced, the tree key (and hence the stage 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. □

Tightness of the security reduction.

As pointed out in [2], a limitation of conventional game hopping proofs for AKE-like protocols is that they do not provide tight reductions. The underlying reason is that the reductions depend on guessing the specific party and session under attack. In the case of a protocol with potentially huge amounts of sessions and users, this leads to an extremely non-tight reduction. While [2] develops some new protocols with tight reductions, their protocols are non-standard in their setup and assumptions. In particular, there is currently no known technique for constructing a tight reduction that is applicable to the ART protocol. Nevertheless, even loose bounds are generally considered useful to increase confidence in the security mechanisms [17, 24].

6.3 Analysis: Authenticated Protocol

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 whose public key was used for setup can derive the group key. We now give an example of such a construction, and analyse its authentication property with Tamarin.

We use X3DH extended with the static-static DH key as our one-round key exchange protocol: agents A and B with long term keys g^a and g^b and ephemeral keys g^x and g^y derive a shared key $K = H(g^{ay}, g^{bx}, g^{xy}, g^{ab})$. Including g^{ab} means that knowing y is not sufficient to impersonate any party to B: an adversary must also know b. To model the authentication property we abstract out the tree construction and replace it with a symbolic "oracle", assigning to any set of public keys a fresh term representing the group key they induce. Anyone may query this oracle if they know one of the corresponding secret keys.

We use Tamarin for mechanised verification. Roughly, we model a protocol role Alice who accepts initial key exchange messages representing new group members, adding the derived keys to her state. At any point she may stop accepting new members and derive a group key via 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 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 is secret, if none of the agents they believe to be in the group have been compromised. The verification is unbounded, allowing an arbitrary number of parties, instances, and group members. The verification of this security property proceeds automatically using several helper lemmas, and takes $\sim$ 15m on a modern desktop.

7 ART Implementation

We implemented the ART protocol described in Section 5 in Java, with source code available at the URL [41]. Our goal is to demonstrate that ART is practical and efficient for groups of a realistic size. *Implementation details are in Appendix D.*

We compare directly to DH ratcheting with pairwise connections, noting that hash ratcheting could be added to ART for a full comparison against Signal's Double Ratchet. We do not benchmark against the sender key design, because it does not achieve PCS.

For our benchmarks, we construct a simple protocol in two phases around both approaches. The first "setup" phase constructs a group, such that at its conclusion any member can send a message. The second "encryption" phase performs an asymmetric update and then encrypts a random message to the entire group. The ART instantiation of the encryption phase comprises an update message and then a single encryption under the group key. The pairwise-channel instantiation follows the Double Ratchet algorithm, sending an update over each channel if the messaging direction changes, and encrypting the message with the latest message key for that channel. We use 32-byte messages, since this is enough space to store an AES-256 data encapsulation key.

We measure wall-clock time and network bandwidth consumption in various scenarios, but our primary metric is the *per-person time/bandwidth cost to send a message*. This is the main cost which is directly and repeatedly visible to users: setup costs, while also important, are only incurred once, while this cost is incurred each time a user sends a message. All data are from a 2016 MacBook Pro with a 3.3 GHz Intel Core i7 and 16 GB of RAM.

7.1 Evaluation

Our results demonstrate that ART is practical for reasonably-sized groups, with setup and sending both taking a few milliseconds for groups of size ten and on the order of one second for groups of size 1000. ART's performance compares favourably to that of pairwise DH ratcheting, as seen in Figure 7a. This is due to server-side fanout: ART allows for broadcasting the same (logarithmic) quantity of data to all peers, while pairwise channels require sending different constant-sized data to everyone.

Depending on how the broadcast is implemented, this yields slightly different benefits. In practice, for the messaging context, broadcast is typically offloaded to server-side fanout. In this case, the *total* number of bytes transmitted in a system using ART is actually larger than for pairwise connections, and recipients must perform a logarithmic instead of constant-size computation[5]. However, in the messaging context, the total amount of data sent is often less important than the message sending latency, which is directly proportional to the amount of data each agent needs to send. Because ART allows for server-side fanout, distributing the sending cost across all parties, it thus allows for significantly lower sending latency than pairwise.

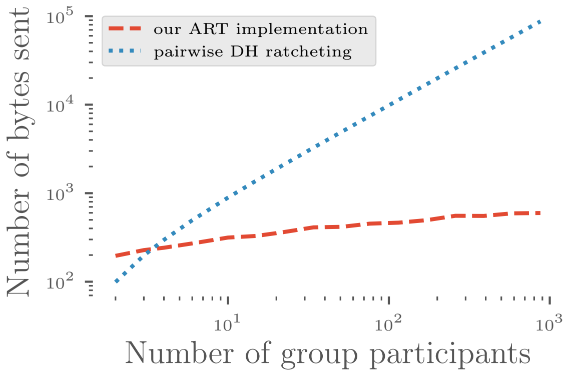
As seen in Figure 7b, the setup costs of ART and pairwise channels are comparable, with the former consistently slightly slower than the latter but with the same asymptotic trend. However, we do remark that pairwise channels spread a quadratic computational effort evenly across all group members, while ART requires the creator to perform a linear amount of work and the responders to perform a logarithmic amount. Although this overhead is minimal for the group sizes normally seen in messaging applications, for large-scale use cases the shared quadratic effort may be a significant performance constraint.

8 Extensions

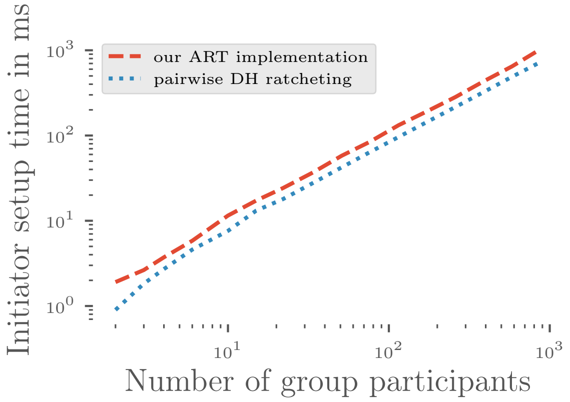
We here remark on various possible extensions to our ART design. In general, because we use standard tree-DH techniques, much of the existing literature is directly applicable. This means that we can directly apply well-studied techniques which do not require interactive communication rounds.

Sender-specific authentication As early as 1999, Wallner, Harder, and Agee [54] 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

[5] To send a single ART message, *every* recipient receives and processes a copy of a logarithmic quantity of data, while over pairwise channels they receive only one constant-sized message.



(a) Outgoing bytes needed to send a 32-byte message.



(b) Total wall-clock setup time to create a group.

Figure 7: Graphs showing metrics from our ART implementation, compared against pairwise DH ratcheting and averaged over four runs. We conjecture that the variance is due to the Java JIT compiler.

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, by rotating and chaining signature keys, we conjecture that it is possible to achieve this authentication post-compromise.

Dynamic groups We refer the reader to e.g. [28] 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. Using "subtrees" in this way allows a user to publish the public key of their subtree as an ephemeral prekey, enabling all their devices to be added to new groups as a single unit. Moreover, users do not need to reveal which device in a subtree triggers an update, thus improving their privacy guarantees.

Chain keys Signal introduced the concept of *chain keys* to support out-of-order message receipt and 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 from a hash chain. The shared secret derived by our GKE can be directly used in the same way, for the same benefits.

9 Conclusion

In this paper, we combined techniques from synchronous group messaging with strong modern security guarantees from asynchronous messaging. Our resulting Asynchronous Ratcheting Trees (ART) design combines the bandwidth benefits of group messaging with the strong security guarantees of modern point-to-point protocols. Our design is the first to show that post-compromise security is efficiently achievable for group messaging as well as pairwise. 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.

ART has seen widespread interest from industry, and forms the basis of two draft RFCs as well as the IETF's MLS working group which has adopted it as a starting point. We hope that it will lead to designs for secure messaging systems which can improve the guarantees provided to users everywhere.

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.

Acknowledgements The authors would like to thank Richard Barnes for pointing out an error in a previous version of the algorithm, and MLS's many contributors for helpful discussions and insights.

References

[1] 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, editors. Volume 6055. Springer, Heidelberg, (May 2010), 351–368.

[2] Christoph Bader, Dennis Hofheinz, Tibor Jager, Eike Kiltz, and Yong Li. 2015. Tightly-secure authenticated key exchange. In *TCC 2015, Part I* (LNCS). Yevgeniy Dodis and Jesper Buus Nielsen, editors. Volume 9014. Springer, Heidelberg, (March 2015), 629–658. DOI: 10.1007/978-3-662-46494-6_26.

[3] Daniel J. Bernstein. 2006. Curve25519: new Diffie-Hellman speed records. In *PKC 2006* (LNCS). Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors. Volume 3958. Springer, Heidelberg, (April 2006), 207–228. DOI: 10.1007/11745853_14.

[4] 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*. Contemporary Mathematics. Volume 324. Caroline Grant Mellesand Jean-Paul Brasseletand Gary Kennedyand Kristin Lauter and Lee McEwan, editors. American Mathematical Society.

[5] 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, editors. Volume 8616. Springer, Heidelberg, (August 2014), 480–499. DOI: 10.1007/978-3-662-44371-2_27.

[6] 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. DOI: 10.1145/1029179.1029200.

[7] 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, editors. Volume 5888. Springer, Heidelberg, (December 2009), 478–497.

[8] Jacqueline Brendel, Marc Fischlin, Felix Günther, and Christian Janson. 2017. Prf-odh: relations, instantiations, and impossibility results. Cryptology ePrint Archive, Report 2017/517. http://eprint.iacr.org/2017/517. (2017).

[9] Emmanuel Bresson, Olivier Chevassut, David Pointcheval, and Jean-Jacques Quisquater. 2001. Provably authenticated group Diffie-Hellman key exchange. In *ACM CCS 2001*. Michael K. Reiter and Pierangela Samarati, editors. ACM Press, (November 2001), 255–264. DOI: 10.1145/501983.502018.

[10] Christina Brzuska, Marc Fischlin, Bogdan Warinschi, and Stephen C. Williams. 2011. Composability of Bellare-Rogaway key exchange protocols. In *ACM CCS 2011*. Yan Chen, George Danezis, and Vitaly Shmatikov, editors. ACM Press, (October 2011), 51–62. DOI: 10.1145/2046707.2046716.

[11] Christian Cachin and Reto Strobl. 2004. Asynchronous group key exchange with failures. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing* (PODC '04). ACM, 357–366. DOI: 10.1145/1011767.1011820.

[12] 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. DOI: 10.1016/j.comcom.2016.08.001.

[13] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. 2016. A formal security analysis of the signal messaging protocol. Cryptology ePrint Archive, Report 2016/1013. http://eprint.iacr.org/2016/1013. (2016).

[14] 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.

[15] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. 2018. On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. ACM, 1802–1819.

[16] 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, editors. Volume 7459. Springer, Heidelberg, (September 2012), 734–751. DOI: 10.1007/978-3-642-33167-1_42.

[17] Ivan Damgård. 2007. A "proof-reading" of some issues in cryptography (invited lecture). In *ICALP 2007* (LNCS). Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors. Volume 4596. Springer, Heidelberg, (July 2007), 2–11. DOI: 10.1007/978-3-540-73420-8_2.

[18] 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, editors. Volume 4886. Springer, Heidelberg, (February 2007), 104–118.

[19] eQualit.ie. 2016. (N+1)sec. (2016). https://learn.equalit.ie/wiki/Np1sec.

[20] Facebook. 2017. Messenger Secret Conversations (Technical Whitepaper Version 2.0). Technical report. Retrieved 05/2017 from https://fbnewsroomus.files.wordpress.com/2016/07/messenger-secret-conversations-technical-whitepaper.pdf.

[21] Michael Farb, Yue-Hsun Lin, Tiffany Hyun-Jin Kim, Jonathan McCune, and Adrian Perrig. 2013. Safeslinger: easy-to-use and secure public-key exchange. In *Proceedings of the 19th Annual International Conference on Mobile Computing and Networking* (MobiCom '13). ACM, 417–428. DOI: 10.1145/2500423.2500428.

[22] 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.

[23] Ian Goldberg, Berkant Ustaoglu, Matthew Van Gundy, and Hao Chen. 2009. Multi-party off-the-record messaging. In *ACM CCS 2009*. Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors. ACM Press, (November 2009), 358–368. DOI: 10.1145/1653662.1653705.

[24] Oded Goldreich. 1997. On the foundations of modern cryptography (invited lecture). In *CRYPTO'97* (LNCS). Burton S. Kaliski Jr., editor. Volume 1294. Springer, Heidelberg, (August 1997), 46–74. DOI: 10.1007/BFb0052227.

[25] 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, (May 2015), 305–320. DOI: 10.1109/SP.2015.26.

[26] internet.org. 2016. State of connectivity 2015. Annual Report. (2016). Retrieved 05/2017 from https://fbnewsroomus.files.wordpress.com/2016/02/state-of-connectivity-2015-2016-02-21-final.pdf.

[27] Antoine Joux. 2004. A one round protocol for tripartite Diffie-Hellman. *Journal of Cryptology*, 17, 4, (September 2004), 263–276. DOI: 10.1007/s00145-004-0312-y.

[28] 2001. *Communication-efficient group key agreement. Trusted Information: The New Decade Challenge.* Springer US. DOI: 10.1007/0-306-46998-7_16.

[29] Yongdae Kim, Adrian Perrig, and Gene Tsudik. 2001. Communication-efficient group key agreement. In *International Federation for Information Processing (IFIP SEC)*. Paris, France, (June 2001).

[30] 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. DOI: 10.1145/352600.352638.

[31] Yongdae Kim, Adrian Perrig, and Gene Tsudik. 2000. Simple and fault-tolerant key agreement for dynamic collaborative groups. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 235–244.

[32] Yongdae Kim, Adrian Perrig, and Gene Tsudik. 2004. Tree-based group key agreement. *ACM Trans. Inf. Syst. Secur.*, (February 2004). DOI: 10.1145/984334.984337.

[33] 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)*.

[34] 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, editors. Volume 4784. Springer, Heidelberg, (November 2007), 1–16.

[35] 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, editors. Volume 2846. Springer, Heidelberg, (October 2003), 357–371. DOI: 10.1007/978-3-540-45203-4_28.

[36] 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. http://eprint.iacr.org/2017/152. (2017).

[37] Moxie Marlinspike. 2013. Forward secrecy for asynchronous messages. Blog. (August 22, 2013). Retrieved 05/2017 from https://whispersystems.org/blog/asynchronous-security/.

[38] Moxie Marlinspike. 2016. Signal protocol documentation. (2016). Retrieved 05/2017 from https://whispersystems.org/docs/.

[39] Moxie Marlinspike. 2016. The x3dh key agreement protocol. Trevor Perrin, editor. (November 2016). Retrieved 11/2017 from https://signal.org/docs/specifications/x3dh/x3dh.pdf.

[40] Ghita Mezzour, Ahren Studer, Michael Farb, Jason Lee, Jonathan McCune, Hsu-Chun Hsiao, and Adrian Perrig. 2010. Ho-Po Key: Leveraging Physical Constraints on Human Motion to Authentically Exchange Information in a Group. Technical report. Carnegie Mellon University, (December 2010).

[41] Jon Millican. 2018. ART prototype implementation. (2018). https://github.com/facebookresearch/asynchronousratchetingtree.

[42] MLS Working Group Chairs. 2018. Messaging layer security working group. https://mlswg.github.io.

[43] Open Whisper Systems. 2014. Libsignal-service-java. (2014). https://github.com/signalapp/libsignal-service-java/blob/c8d7c3c00445a81b81e0a7305151cda4534ba299/java/src/main/java/org/whispersystems/signalservice/api/SignalServiceMessageSender.java#L497.

[44] Adrian Perrig. 1999. Efficient collaborative key management protocols for secure autonomous group communication. In *Proceedings of International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC)*. (July 1999), 192–202.

[45] Adrian Perrig, Dawn Song, and Doug Tygar. 2001. ELK, a new protocol for efficient large-group key distribution. In *Proceedings of IEEE Symposium on Security and Privacy*. (May 2001).

[46] Paul Rösler, Christian Mainka, and Jörg Schwenk. 2018. More is less: on the end-to-end security of group chats in signal, whatsapp, and threema. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*, 415–429. DOI: 10.1109/EuroSP.2018.00036.

[47] Benedikt Schmidt, Simon Meier, Cas 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. DOI: 10.1109/CSF.2012.25.

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

[49] Dmitry Skiba. 2008. Trevorbernard/curve25519-java. GitHub repository. (February 23, 2008). Retrieved 05/2017 from https://github.com/trevorbernard/curve25519-java.

[50] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable Cross-Language Services Implementation. Technical report. Retrieved 11/2017 from https://thrift.apache.org/static/files/thrift-20070401.pdf.

[51] 1990. *A secure audio teleconference system. Advances in Cryptology — CRYPTO' 88: Proceedings*. Springer New York. DOI: 10.1007/0-387-34799-2_37.

[52] Michael Steiner, Gene Tsudik, and Michael Waidner. 2000. Key agreement in dynamic peer groups. *IEEE Transactions on Parallel and Distributed Systems*, 11, 8, (August 2000), 769–780.

[53] The Guardian. 2017. Contact the guardian securely. (2017). Retrieved 06/2017 from https://gu.com/tip-us-off.

[54] D. Wallner, E. Harder, and R. Agee. 1999. Key management for multicast: issues and architectures. RFC. United States, (1999).

[55] WhatsApp. 2016. WhatsApp Encryption Overview. Technical report. Retrieved 07/2016 from https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf.

[56] Chung Kei Wong, Mohamed Gouda, and Simon S. Lam. 2000. Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8, 1, (February 2000), 16–30.

[57] 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*, 1–18. DOI: 10.1007/s10207-017-0373-1.

A Pseudocode Protocol Definitions

We include in Figure 8 precise pseudocode for the ART algorithms, including both "real" state π corresponding to data which might be kept in the memory of an implementation, and bookkeeping state σ which is used in the definition of the security game.

B Queries in the Computational Model

We give formal definitions of the adversary queries used in our computational model in Table 2.

C Computational Security Proof

Remark 4 (Conditions on ι). The property we need of ι is similar to the DDH assumption: given randomly sampled g^x and g^y, it should be computationally hard to distinguish $\iota(g^{xy})$ from $\iota(g^{zy})$ where g^z is uniformly random. This is a $\mathsf{PRF\text{-}ODH}$ assumption without an oracle [8]. This property is commonly satisfied. For example, if we assume that $\iota()$ is a random oracle or even the identity function, then this property holds under the DDH assumption.

Remark 5. Instead of encoding authentication as a restriction on the sessions against which it is valid to issue a $\mathsf{RevSessKey}$ query, we have separated it out into another security experiment: the indistinguishability experiment simply prohibits any $\mathsf{RevSessKey}$ query which would reveal the key of the Tested session, and the new experiment shows that only sessions which "should" derive the same key as the Tested session in fact do.

We call the new experiment the partnering experiment, and its definition follows.

Algorithm 1 Asynchronous group setup

```
procedure SetupGroup($(IK_i, EK_i)_{i=1}^{n-1}$)
    // set up a group with $n-1$ identity and ephemeral keys of peers
    $\pi.\lambda = \lambda_0 \overset{\$}{:=} \mathrm{DHKeyGen}()$
    $suk \overset{\$}{:=} \mathrm{KeyExchangeKeyGen}()$
    for $i := 1 \ldots n-1$ do // generate leaf keys for each agent
        $\lambda_i := \iota\big(\textsc{KeyExchange}(\pi.ik, IK_i, suk, EK_i)\big)$
        add $(\lambda_i, \mathit{sid}(\pi, \sigma))$ to $\sigma$.HonestKeys
    $T_{\text{secret}} := \textsc{CreateTree}(\lambda_0, \lambda_1, \ldots, \lambda_{n_{\text{peers}}})$
    $\pi.T := \textsc{PublicKeys}(T_{\text{secret}})$
    $\pi.\mathrm{IDs} := g^{\pi.ik} \| IK_1 \| \ldots \| IK_{n_{\text{peers}}}$
    $\pi.\mathrm{EKs} := g^{\pi.ek} \| EK_1 \| \ldots \| EK_{n_{\text{peers}}}$
    $x := \pi.\mathrm{IDs}, \pi.\mathrm{EKs}, SUK, \pi.T$
    $m := (x, \textsc{Sign}(x; \pi.ik))$
    $\pi.tk := (T_{\text{secret}})_{0,0}$
    $\pi.\mathrm{idx} := 0$
    $\sigma.\ell[\pi.\mathrm{idx}] := 1$
    $\sigma.\ell[x] := 0$ for $0 < x \leq n$
    $\pi.\bar{P} := \textsc{Copath}(T, 0)$
    $\pi.sk = 0$
    DeriveStageKey()
    return $m$
```

Algorithm 2 Helper functions

```
function LeftSubtreeSize($x$)
    // height of the left subtree if there are $x$ elements
    return $2^{\lceil \log_2(x) \rceil - 1}$

function CreateTree($\lambda_0, \lambda_1, \ldots, \lambda_n$) // tree with $n$ leaves
    if $n = 0$ then return $(\text{leaf}, \lambda_0)$
    $h := \textsc{LeftSubtreeSize}(n)$
    $(L, lk) := \textsc{CreateTree}(\lambda_0, \ldots, \lambda_{(h-1)})$ // complete left subtree
    $(R, rk) := \textsc{CreateTree}(\lambda_h, \ldots, \lambda_{n-1})$ // right subtree
    $k := \iota\big(g^{(lk)(rk)}\big)$
    return $(\text{node}((L, lk), (R, rk)), k)$

function PublicKeys($T := \text{node}(L, R), k$)
    if $T = \emptyset$ then return $\emptyset$
    return $\text{node}(\textsc{PublicKeys}(L), \textsc{PublicKeys}(R)), g^k$

function Copath($T, i$) // where $i$ is the index of the leaf and $|T| = \#$leaves
    if $i < \textsc{LeftSubtreeSize}(|T|)$ then // $i$ is in the complete left subtree
        return $g^{T_{1,1}} \| \textsc{Copath}(T_{1,0}, i)$
    else// $i$ is in the possibly incomplete right subtree
        return $g^{T_{1,0}} \| \textsc{Copath}(T_{1,1}, i - 2^l)$

function PathNodeKeys($\lambda, \bar{P}$) // leaf key and the copath of public keys
    $nks_{|\bar{P}|} := \lambda$
    for $j := (|\bar{P}| - 1) \ldots 1$ do
        $nks_j := \iota\big((\bar{P}_j)^{nks_{j+1}}\big)$
    return $(\bar{P}_0)^{nks_1} \| nks_1 \| \ldots \| nks_{|\bar{P}|}$

function DeriveStageKey
    $\pi.sk := \mathrm{KDF}(\pi.sk, \pi.tk, \pi.\mathrm{IDs}, \pi.T)$
    return
```

Algorithm 3 Receiving a setup message as agent at index i

```
procedure ProcessSetupMessage($m$)
    $(x, s) := m$
    assert $\textsc{SigVerify}(x, s, (x.\mathrm{IDs})_0)$
    $(\pi.\mathrm{IDs}, \pi.T) := (x.\mathrm{IDs}, x.T)$ // store agent ids and copath in state
    $ek$ := ephemeral prekey corresponding to $EK$ from $\pi$
    $\pi.\lambda := \iota\big(\textsc{KeyExchange}(\pi.ik, (\pi.\mathrm{IDs})_0, ek, x.SUK)\big)$
    $nks := \textsc{PathNodeKeys}(\pi.\lambda, \pi.\bar{P})$
    $\pi.tk := nks_0$ // store initial tree key
    $\pi.\mathrm{idx} = i$
    $\sigma.\ell[0] := 1$
    $\sigma.\ell[x] := 0$ for $0 < x \leq n$
    $\pi.sk = 0$
    DeriveStageKey()
    return
```

Algorithm 4 Agent updating their key

```
procedure UpdateKey
    $\pi.\lambda \overset{\$}{:=} \mathrm{DHKeyGen}()$
    $\sigma.\ell[\pi.\mathrm{idx}] := \sigma.\ell[\pi.\mathrm{idx}] + 1$
    add $(\pi.\lambda, \mathit{sid}(\pi, \sigma))$ to $\sigma$.HonestKeys
    $nks := \textsc{PathNodeKeys}(\lambda, \pi.\bar{P})$
    $x := \pi.\mathrm{idx} \| g^{nks_1} \| \ldots \| g^{nks_{|\bar{P}|}}$
    $\pi.tk := nks_0$
    $m := (x, \mathrm{MAC}(x; \mathrm{KDF}('mac', \pi.sk)))$
    $\sigma.t := \sigma.t + 1$
    DeriveStageKey()
    return $m$
```

Algorithm 5 Processing another agent's key update

```
procedure ProcessUpdateMessage($m$)
    $(x, \mu) := m$
    assert $\mathrm{MAC}\textsc{Verify}(x, \mu, \mathrm{KDF}('mac', \pi.sk))$
    $j, \mathrm{nks} := x$
    $\nu := \textsc{IndexToUpdate}(\lceil \log_2 n \rceil, 0, \pi.\mathrm{idx}, j)$
    $\pi.\bar{P}_\nu := U_\nu$ // index $\nu$ of the copath has been updated in this message
    $nks := \textsc{PathNodeKeys}(\pi.\lambda, \pi.\bar{P})$
    $\pi.tk := nks_0$
    $\sigma.\ell[j] := \sigma.\ell[j] + 1$
    $\sigma.t := \sigma.t + 1$
    DeriveStageKey()
    return

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

Figure 8: Pseudocode descriptions of the algorithms in our ART design. Informal explanations can be found in Section 5. **Procedure** denotes subroutines which are used by the protocol algorithms PSend/PRecv and **function** denotes ones which are not. Procedures operate on and mutate the agent's current state π and σ, receive an optional input message from the adversary, and return an optional output message to it. When sending a tuple, we implicitly uniquely encode it as a bitstring (to avoid type confusion errors), and when receiving one we uniquely decode it. KeyExchange derives a shared key from secret/public identity/ephemeral keys, i.e. $\textsc{KeyExchange}(ik_A, IK_B, ek_A, EK_b) = \textsc{KeyExchange}(IK_A, ik_B, EK_A, ek_B)$.

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, $(\mathit{Alice}, 3, 4)$ identifies the fourth stage of Alice's third session. We use m for messages and b, b' for bits.

$\mathsf{Create}(u, v_1, v_2, \dots v_{n-1})$	Given a set of intended peers $v_1, \dots, v_{n-1}$ $(n \leq \gamma)$, the challenger executes Activate to prepare a new state π, prepares a new bookkeeping state σ with $\sigma.i$ set to the number of times $\mathsf{Create}(u, \dots)$ has already been called, and initialises a new role oracle with states π and σ for agent u.
$\mathsf{ASend}(u, i, m)$	Given a message m and a session (u, i) with state π, execute $\pi' := \mathrm{PRecv}(\pi, m)$ and set the session state to π'. u must be a valid agent identifier and $\mathsf{Create}(u, \dots)$ must have been called at least $i - 1$ times. This query models sending a message to a session.
$\mathsf{ARecv}(u, i, d)$	Given a session (u, i) with state π, execute $\pi', m := \mathrm{PSend}(\pi, d)$, update the session state to π' and return the message m. This query models a role oracle performing of the protocol's actions.
$\mathsf{RevSessKey}(u, i, t)$	Given (u, i, t), return $\pi.sk$ where π is the stage with $\mathit{sid}(\pi) = (u, i, t)$ if it exists. This query models 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 (u, i). This query models the corruption of an agent, either in their initial key generation (at $t = 0$) or afterwards $(t > 0)$.
$\mathsf{Test}(u, i, t)$	Given (u, i, t), let k_0 denote the key computed by user u at stage t of session (u, i), and let k_1 denote a uniformly randomly sampled key from the challenger. The challenger flips a coin $b \overset{\$}{:=} \mathrm{Uniform}(\{0, 1\})$ and returns k_b.
$\mathsf{Guess}(b')$	The adversary immediately terminates its execution after this query.

Definition 9 (Partnering experiment)**.** We define the partnering experiment as follows. At the start of the game, the challenger initialises all parties as in the security experiment. The adversary then asks a series of Create, ASend or ARecv queries, and eventually terminates. There is no additional model state and no other queries are permitted.

When the game ends, the adversary wins if and only if for any session (u, i, t) with $(u, i, t).\sigma.\mathrm{status} = \mathtt{accept}$, any of the following hold.

(i) disagreement on group members: there exists another stage (v, j, s) deriving the same key as (u, i, t) but with $(u, i, t).\mathrm{IDs} \neq (v, j, s).\mathrm{IDs}$

(ii) incorrect peer: there exists a stage (v, j, s) deriving the same key as (u, i, t) with $v \neq u$ and $v \notin (u, i, t).\mathrm{IDs}$

(iii) repeated session key: there exists another session (u, i', t'), $i' \neq i$, deriving the same key as (u, i, t)

(iv) too many copies of a peer: for any peer identity v appearing $n > 0$ times in $(u, i, t).\mathrm{IDs}$, $v \neq u$, there exist $n + 1$ stages $(v, \cdot, \cdot)$ deriving the same key as (u, i, t)

Theorem 2. *In ART, when the KDF is modelled as a random oracle, the probability that any PPT adversary wins the partnering game is negligible in the security parameter.*

Proof sketch. The result follows directly from the fact that $\pi.\mathrm{IDs}$ is an argument to the KDF when deriving stage keys. If the KDF is a random oracle its output values do not collide, and thus equal output values imply equal input values, which is enough to rule out the cases in the partnering security experiment.

Suppose there exists an adversary $\mathcal{A}$ which wins the partnering security game. By definition, it wins if one of the four cases occurs and we consider each one in turn.

First, suppose that it wins because there exist two stages (u, i, t) and (v, j, s) deriving the same key but with (u, i, t).IDs $\neq$ (v, j, s).IDs. The stage key is derived as $\pi.sk :=$ KDF$(\pi.sk, \pi.tk, \pi.\text{IDs}, \pi.T)$. In particular, equality of stage keys implies equality of IDs (except with negligible probability of collisions in the random oracle), so this case is impossible.

Second, suppose that it wins because there exists a stage (v, j, s) deriving the same key as (u, i, t) but with $v \notin (u, i, t)$.IDs. As in the first case, we know that (u, i, t).IDs $=$ (v, j, s).IDs, and thus $v \notin (v, j, s)$.IDs. However, this contradicts the fact that agents always believe they are in their own groups, so this case is impossible.

Third, suppose that there exist two sessions (u, i, t) and (u, i', t') deriving the same key. Recall that each stage derives an ephemeral key, and each stage's own ephemeral key is included in its local key derivation. For the derived keys to be equal, therefore, the ephemeral keys generated by both agents would have to be equal as well, which would require a DH collision. This happens only with negligible probability (formally, we make a game hop to a game which aborts if there is a DH collision, and bound the difference between the games; this argument appears in the proof sketch below), and hence this case is impossible unless $i = i'$.

Fourth and finally, suppose that it wins because there exist $n + 1$ stages $(v, \cdot, \cdot)$ deriving the same key as (u, i, t) while there are only n copies of v in (u, i, t).IDs. Since there are only n copies of v in (u, i, t).IDs, either

(i) one of the $n + 1$ must have v.idx not equal to an index of v in (u, i, t).IDs, or

(ii) two of the stages $(v, \cdot, \cdot)$ must "collide", having the same v.idx.

In the first case, the disagreement implies that (u, i, t).IDs $\neq$ (v, j, s).IDs and hence that the derived keys are distinct, which is a contradiction. In the second case we again use uniqueness of ephemeral keys: the two colliding stages must have derived distinct ephemeral keys, at most one of which is the key appearing (u, i, t).IDs, and hence the stages must derive distinct keys.

We have ruled out all cases, and thus are done. □

Theorem 1. *Let $\mathsf{n_P}$, $\mathsf{n_S}$ and $\mathsf{n_T}$ denote bounds on the number of parties, sessions and stages in the security experiment respectively. Under the* PRF-ODH *assumption with KDFs modeled as random oracles, the success probability of any PPT adversary against the security experiment for our protocol is bounded above by*

$$\frac{1}{2} + \frac{\binom{\mathsf{n_P n_S n_T}}{2}}{q} + \gamma(\mathsf{n_P n_S n_T}^2)^\gamma\left(\epsilon_{\mathsf{PRF\text{-}ODH}} + {}^1\!/_q\right) + \mathsf{negl}(\lambda)$$

where $\epsilon_{\mathsf{PRF\text{-}ODH}}$ bounds the advantage of a PPT adversary against the PRF-ODH *game. (This bound depends only on $\epsilon_{\mathsf{PRF\text{-}ODH}}$ and not* KeyExchange *because it is unauthenticated.)*

Proof. Security in this sense means that no efficient adversary can break the key indistinguishability game against the 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 with non-negligible advantage above $^1\!/_2$.

By the definition of freshness, (u, i, t) is fresh (Definition 7) precisely 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 sk, and

(iv) one of the following criteria holds:

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

   (b) the current copath is fresh.

The proof is a case distinction based on adversarial behaviour. We will also construct a sequence of related games as per the game hopping proof technique [48]. Let Game 0 denote the game from the original security experiment. 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.

Recall that due to technical limitations of key indistinguishability models we are unable to faithfully model the explicit MACs which ART uses in group creation and key update messages. Instead, for the remainder of the proof we omit them from the protocol, and specify authentication "by fiat" through our freshness predicate—that is, we rule out attacks in which the authentication of these messages is violated.

At any point in a run of the game, by construction such a tuple (u,i,t) uniquely identifies a corresponding pair of states π and σ if they exist (Definition 3). To simplify our notation, therefore, where is it more convenient we refer to session and bookkeeping states directly by their identifiers, so for example by $(u,i,t).\pi.x$ we mean $\pi.x$ of (u,i,t) and by $(u,i,t).\sigma.y$ we mean $\sigma.y$ of (u,i,t).

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 $\mathsf{n_S n_T}$ ephemeral DH keys generated per party. There are therefore a total maximum of $\mathsf{n_P n_S n_T}$ DH keys, each pair of which must not collide. All keys are generated in the same DH group of order q so each of the $\binom{\mathsf{n_P n_S n_T}}{2}$ pairs has probability $1/q$ of colliding. Therefore, we have the following bound:

$$\mathsf{Adv}_0 \leq \frac{\binom{\mathsf{n_P n_S n_T}}{2}}{q} + \mathsf{Adv}_1$$

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.

Additionally, the challenger guesses a corresponding key counter value ℓ' and aborts if $\ell' \neq (u,i,t).\sigma.\ell[(u,i,t).\pi.\mathrm{idx}]$. In other words, the challenger also attempts to guess the number of sent DH keys from the Test. There are at most $\mathsf{n_T}$ possible sent keys.

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

$$\mathsf{Adv}_1 \leq \mathsf{n_P n_S n_T}^2 \ \mathsf{Adv}_2$$

Game 3. In this game, the challenger guesses in advance the peer sessions associated with each leaf key in $(u, i, t).\pi.T$ (if they exist), and aborts if both of the following two conditions are met: (i) they are not unique and (ii) the non-unique sessions have contributed their own leaf key.

Precisely, the challenger does the following: for each leaf l in $(u, i, t).\pi.T$, it guesses a triple of indices $(v'_l, j'_l, s'_l) \in [n_{\mathsf{n_P}}] \times [\mathsf{n_S}] \times [\mathsf{n_T}]$ and aborts if there exists a session (v, j, s) with $(v, j, s).\pi.\text{idx} = l$ and $(v, j, s).\pi.T = (u, i, t).\pi.T$ and $(v, j, s).\sigma.\ell[(v, j, s).\pi.\text{idx}] > 0$ but $(v'_l, j'_l, s'_l) \neq (v, j, s)$. In other words, for each leaf l in the tree of the Test session, the challenger tries to guess in advance the agent, agent's session, and stage of the session, that have the same DH tree in session memory contents that the Test session (u, i, t) has, and believe that their leaf key is at leaf l, where the peers are no longer using a setup key from u, and aborts if any are not unique. Note that it might be the case that no such (v_l, j_l, s_l) exist, but this game ensures that if they do exist, they are uniquely defined and known in advance by the challenger.

Recall that γ denotes the maximum group size. From $(u, i, t).\pi.T$ we can derive an ordered list of the peers associated with each leaf at stage t. Therefore, there are no more than $\gamma - 1$ such leafs, so we will assume the worst case of making $\gamma - 1$ guesses.

Uniqueness of the guessed tuples follows from the fact that in Game 1 we ensured in advance that honestly generated DH values are unique: the challenger guesses sessions that could possibly have the same view of the internal tree structure as the Test session. This means (without loss of generality) that Bob is at leaf 1, Charlie at leaf 2, etc. For uniqueness of the guessed sessions with the same view of the internal tree structure as the Test not to hold, this must mean at least two sessions with the same internal view at a particular leaf. To have the same view, they must have the same session actor identity. Also, we only abort if $(v_l, j_l, s_l).\sigma.t > 0$. This means that for uniqueness not to hold, the same actor must have generated the same DH value at the leaf l. But this cannot happen by Game 1.

Additionally, for each leaf l, the challenger guesses a corresponding key counter value l_c and aborts if $(u, i, t).\sigma.\ell[l] \neq l_c$. In other words, the challenger also attempts to guess the number of received DH keys from each node l in the Test. There are at most $\mathsf{n_T}$ possible guesses for each leaf.

The guesses are made uniformly randomly before the game starts. This therefore provides the following bound:

$$\mathsf{Adv}_2 \leq (\mathsf{n_P n_S n_T}^2)^{\gamma-1} \ \mathsf{Adv}_3$$

Case distinction. At this point in the proof, we do a case distinction based on adversary behaviour. Consider the event E defined to be true when the current copath of u at $(u, i, t).\pi.T$ 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. Our game hopping sequence splits: we either proceed from case (i) game 4, 5, 6..., or case (ii) game 4, 5, 6...

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 Λ, where each $\lambda_j \in \Lambda$ was output by an honest stage against which no $\mathsf{RevRandom}$ query was issued. Without loss of generality, we define λ_1 to be the leaf key of u in $(u, i, t).\pi.T$.

Case (i), Game 4.

Recall that the parent of the first two leaf nodes, λ_1 and λ_2, is defined as $g^{\lambda_1\lambda_2}$. The key derived from this is defined as $\iota(g^{\lambda_1\lambda_2})$. We define a new game in which, in the local stage key computation of the actor of the Test session and stage and any match (which is unique by the previous game), $\iota(g^{\lambda_1\lambda_2})$ is replaced with $\iota(g^z)$ for uniformly randomly chosen DH group exponent z, and all subsequent computations upwards along the path of the tree use $\iota(g^z)$ instead of $\iota(g^{\lambda_1\lambda_2})$.

This is a game hop based on indistinguishability [48]. 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 is indistinguishable from game 3 under the $\mathsf{PRF\text{-}ODH}$ 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 $\mathsf{PRF\text{-}ODH}$ assumption. This implies that $\mathsf{Adv}_4 \leq \mathsf{Adv}_3 + \max_{\mathcal{D}} \epsilon_{\mathcal{D}}$, where $\epsilon_{\mathcal{D}}$ is the probability that a PPT distinguisher $\mathcal{D}$ correctly distinguishes between Games 3 and 4(i).1.

It remains to bound $\epsilon_{\mathcal{D}}$, which we do with a reduction to $\mathsf{PRF\text{-}ODH}$. Specifically, suppose $\mathcal{D}$ is such a distinguisher. We construct an adversary $\mathcal{A}(\mathcal{D})$ against the $\mathsf{PRF\text{-}ODH}$ game as follows: Given $\mathsf{PRF\text{-}ODH}$ challenge $g^x, g^y, \iota(g^z)$ 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 $g^x = g^{x_1}, g^y = g^{x_2}$ and $\iota(g^z) = \iota(g^{x_1x_2})$.

Our constructed $\mathsf{PRF\text{-}ODH}$ adversary is given $\iota(g^z)$, which by construction is the node key at the parent of Alice's and Bob's leaf nodes. It can therefore replace this node key with $\iota(g^z)$ and, using this secret, compute all public DH intermediate keys up the tree that depend on $\iota(g^z)$, including the tree key at the top of the tree. This game is a hybrid game between Game 3 and Game 4, 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. In particular, since this is case (i), the leaf keys are honestly sent and from game 3 the challenger knows which agent's session and stage's they are generated at in advance, as well as which generated DH this will be. In other words, the challenger knows (v, j, s) and the associated counter for how many DH keys have been generated $(v, j, t).\sigma.\ell[(v, j, t).\pi.\text{idx}]$. So if it correctly guesses agent v, session j and stage s without aborting as in Game 3, then instead of honestly answering a $\mathsf{ASend}(v, j, t)$ query when the ℓth DH key is due to be sent in the session (v, j, s) to the Test (or Create query if its the initial DH key) by running the protocol to generate an ephemeral key, the challenger instead inserts the $\mathsf{PRF\text{-}ODH}$ challenge value. This value is unique as there is only one sent per query so the challenger knows where to insert it. Precisely, the challenger does not follow the protocol to honestly generate a DH key, and instead uses the one provided in the $\mathsf{PRF\text{-}ODH}$ game.

Because of the earlier game hops the simulator knows where to inject the replaced values in the simulation, and because of the freshness predicate they are honest. Similarly, because of the freshness predicate it never has to answer a $\mathsf{RevRandom}$ query against either of these two values, and it can honestly simulate any other reveal queries. Therefore the simulation is sound.

In Game 1 we ensured no DH keys collide, and with probability $1/q$ the $\mathsf{PRF\text{-}ODH}$ challenger may provide challenge values $g^x = g^y$, in which case the simulator must abort. Fortunately this happens with negligible probability. Thus, we have the bound:

$$\mathsf{Adv}_3 \leq \mathsf{Adv}_4 + \epsilon_{\mathsf{PRF\text{-}ODH}} + {}^1\!/_q$$

We will now iteratively repeat this game hop for all other fresh DH values in the tree $(u, i, t).\pi.T$. Because we are in case (i) and know from the previous game hops were to insert the PRF-ODH challenge DH values, we will therefore conclude that each node key in turn is indistinguishable from random. Repeating this process, the eventual conclusion will be that the secret at the root of the tree is also indistinguishable from random.

Case (i), Game $4+k$ where $1 \leq k \leq \gamma$. We repeat the replacement performed in the previous game, but for the next pair of sibling nodes. Again, detecting this replacement would require violating PRF-ODH. 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. We iteratively replace DH keys using the PRF-ODH 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 $n \leq \gamma$ such game hops due to our tree structure. Thus

$$\mathsf{Adv}_{\mathsf{n_P}} \leq \gamma\,(\epsilon_{\mathsf{PRF\text{-}ODH}} + {}^1\!/_q) + 0$$

Case (ii), Game 4. We now proceed with case (ii), restarting our game hopping sequence from Game 3. Assume now that E does not hold, and thus the copath in the session state of the Tested stage is not fresh. Since the Tested stage must be fresh, the first disjunct of the final clause of the freshness predicate must hold: that $t > 0$ and the stage with sid $(u, i, t - 1)$ 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 with non-negligible advantage if the tested session has stage number less than or equal to k. 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 queries $\mathsf{Test}(u, i, t)$, then this means stage t must be fresh. Let RO be the event that the adversary queried the random oracle and received the key of the Test stage as a reply.

If RO does not hold, then since since the adversary is not allowed to reveal the key because of the freshness predicate, the only option is for a key replication attack. We can perform a single game hop in which we replace the stage key with a random value. Since the random oracle response is by construction a random value, this replacement is indistinguishable and the resulting advantage for the adversary is zero.

Thus, we conclude that RO must hold. Since random oracle produce collisions with only negligible probability, it must be the case that the adversary queried the KDF on the same input that u did on the stage key computation in the stage with sid (u, i, k). In particular, it must have queried the random oracle on the stage key as that is one of the inputs. This adversary therefore has a distinguishing advantage against the previous stage, (noting that this is case (ii) so it is fresh by definition). This contradicts our induction hypothesis.

Specifically, given such an adversary $\mathcal{A}$ we can 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 $\mathsf{Test}(u, i, k)$ query, $\mathcal{A}'$ issues a $\mathsf{Test}(u, i, k-1)$ 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, the stage with sid $(u, i, k-1)$ is fresh and its stage 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 k with non-negligible probability; the result thus holds in case (ii) by induction. □

D Further Measurement Details

Our prototype implementation is available from [41]. Our algorithm is implemented in the file `ART.java`, with the other files primarily providing the required container and utility classes.

For our Diffie-Hellman group operations we use a Java implementation [49] of Curve25519 [3]. 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. We use HKDF with SHA256 for all key derivations.

We use the X3DH key exchange algorithm for our initial authenticated key exchanges in both algorithms, extended to include the static-static DH key. Encryption keys for messages ("message keys") are taken straight from the stage keys of each implementation, instead of using the "chain keys" approach used in the Double Ratchet algorithm. We made this choice because ART does not include hash ratcheting in its raw form, although we note that this could be added to construct an ART-Double Ratchet.

We pass messages between sessions as byte arrays in memory, to allow us to measure relative network costs without actually transmitting them over a network device. We use the Apache Thrift [50] library and toolchain to serialise messages, to mimic as closely as possible an actual wire format used for RPC.

E Terms and abbreviations

AKE Authenticated Key Exchange
ART Asynchronous Ratcheting Trees
DDH Decisional Diffie-Hellman
DH Diffie-Hellman
iO indistinguishability obfuscation
MAC Message Authentication Code
PCS Post-Compromise Security
PPT probabilistic polynomial-time

F Summary of changes

V1.0, June 5th, 2017

Initial version

V1.1, July 17th, 2017

- Fixed error pointed out by Richard Barnes
- Introduced explicit name (ART) for our design

V2.0, December 5th, 2017

Major changes:

- Reworked and improved computational model
  - Adversary-security game interaction is explicit (via PSend and PRecv queries)
  - Pseudocode algorithms take adversary messages as input and return responses
  - Bookkeeping state σ now tracks implicit protocol variables
  - Clarified bounds in some game hops
- Redesigned Java implementation
  - Implemented pairwise ratchet implementation as a comparison
  - Accurate timing measurements for both including crossover point
  - Graphed various metrics for a range of group sizes
- Improved related work discussion (esp. SafeSlinger) and scoping explanation

V2.1, January 18th, 2018

- Clarified motivation for adding static-static DH key, and removed reference to UKS attacks
- Removed "session key" nomenclature

V2.2, May 15th, 2018

- Rewrote algorithm description and figures more operationally
- Added partnering game to security experiment
- Added reference to IETF working group
- Elaborated on usefulness and relevance of PCS
- Clarified tree notation by writing both secret and public keys
- The Signal mobile app uses pairwise channels, not sender keys

V2.3, March 2nd, 2020

- Fixed typos throughout, minor improvements to writing
- Added reference to CCS 2018 version