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

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

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

Version 2.1, January 18th, 2018*

Abstract

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

One reason for this discrepancy in security guarantees, despite the large body of work on group key agreement, is that most existing protocol designs are fundamentally *synchronous*, and thus cannot be used in the asynchronous world of mobile communications. In this paper we show that this is not necessary, presenting a design for a tree-based group key exchange protocol in which no two parties ever need to be online at the same time, which we call *Asynchronous Ratcheting Tree (ART)*. ART achieves strong security guarantees, in particular including post-compromise security.

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

*A summary of changes is given in Appendix C.

1 Introduction

The level of security offered by secure messaging systems has improved substantially over recent years; for example, WhatsApp now provides end-to-end encryption for its billion active users, based on Open Whisper Systems' Signal Protocol [33, 45], and the Guardian publishes Signal contact details for its investigative journalism teams [20]. An important constraint of modern messaging systems, compared to related protocols such as those used for key exchange, is that they must allow for *asynchronous communication*: Alice must be able to send a message to Bob even if Bob is currently offline. Typically, the encrypted message is temporarily stored on a (possibly untrusted) server, to be delivered to Bob once he comes online again.

Asynchronicity means that standard solutions to achieve perfect forward secrecy (PFS), such as a Diffie–Hellman (DH) key exchange, do not apply directly. This has driven the development of novel techniques to achieve PFS without interaction, e.g., using sets of "prekeys" [32] that Bob uploads to a server, essentially serving as precomputed DH keys, or by using puncturable encryption [19].

Moreover, some modern messaging protocols offer a property called Post-Compromise Security (PCS) [11], sometimes referred to as "future secrecy" or "self-healing". For PCS, even after Alice's device is entirely compromised by an adversary, she may later be able to establish secure communications with others after a single unintercepted exchange. 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. Thus far, PCS-style properties have only been proven for point-to-point protocols [10], and they are only achievable by stateful protocols [11].

In practice however, point-to-point communication does not suffice for real-world messaging applications, in which group and multi-device messaging are often important features. In theory, it is easy to solve this: Alice uses the point-to-point protocol with each of her communication partners. However, as group sizes become larger, this leads to inefficient systems in which the bandwidth and computational cost for sending a message grows linearly with the group size (as each recipient gets their own, differently encrypted, copy of the message). In many real-world scenarios, this inefficiency can be problematic, especially in areas with restricted bandwidth or high data costs (e.g., 2G networks in the developing world). The 2015 State of Connectivity report by `internet.org` [21] lists affordability of mobile data as one of the four major barriers to global connectivity, with a developing-world average monthly data use of just 255 MB/device.

Instead of using a point-to-point protocol with each group member, a theoretical alternative is to use a group protocol [6, 7, 13, 23, 24, 25, 26, 27, 30, 37, 38]. These typically use tree structures based on DH keys to combine the participants' individual keys into a group key. This reduces both the computational effort and bandwidth required to send a message, as the sender sends only one copy of each message encrypted under the group key. However, such protocols are in general not asynchronous, and do not consider PCS—they do not make any guarantees after the adversary completely compromises a participant.

Synchronicity of existing group protocols, among other considerations, means that 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 and Facebook Messenger 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 [15, 45]. 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. new device registration, 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, end-to-end encrypted group messaging solution that

(i) allows participants to communicate *asynchronously*,
(ii) *does not require* effort linear in the size of the group to send a message,
(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 the following:

(1) We design a fully-asynchronous tree-based group key exchange protocol that offers modern strong security properties, called Asynchronous Ratcheting Tree (ART). The ART protocol derives a group key for a set of agents without any pair needing to be online at the same time. Modern messaging protocols must work fully asynchronously, and ART enables this. Notably, ART's properties include PCS: even after total compromise, an agent can participate in a secure group key exchange.

(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 decisional DH problem, and a symbolic verification of its authentication property. Our hybrid argument follows the style of e.g. [28].

(3) We present and evaluate a proof-of-concept Java implementation of all of our core ART algorithms, increasing confidence in the correctness and feasibility of our design.

Our design approach is of independent interest beyond our specific construction. In particular, by using simple and well-studied constructions, our design 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 [36].

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.

2.1 Other Group Messaging Protocols

2.1.1 OTR-style

Goldberg et al. [18] define Multi-Party Off the Record Messaging (mpOTR) as a generalisation of the classic OTR [5] 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 [14]. 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.

2.1.2 Assuming an authentic network

Cachin and Strobl [8] discuss "asynchronous" group key exchange in the setting of distributed systems, in the sense that they do not rely on a centralised clock. Their approach allows certain parties to crash without preventing the protocol from terminating. However, they require several interactive rounds of communication, and do not provide PCS. Steiner, Tsudik, and Waidner [43] also work in this framework.

2.1.3 Physical approaches

Some work uses physical constraints to restrict malicious group members. For example, HoPoKey [35] has its participants arrange themselves into a circle, with neighbours interacting. This allows it to derive strong security properties. Our goal, however, is to allow users to communicate without requiring physical co-location.

2.1.4 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 [45]. 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). Thus, this variant does not provide PCS.

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

2.1.5 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 $g^{x_0}, \dots, g^{x_n}$ 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 [22] gives a pairing-based construction. However, for general n construction of such a primitive is a known open problem. Boneh and Silverberg [3] essentially generalise the Joux protocol with a construction from an $(n-1)$-non-degenerate linear map on the integers. Boneh and Zhandry [4] present one from indistinguishability obfuscation (iO), and recent work by Ma and Zhandry [31] formalises the concept as an "encryptor combiner" and gives constructions from iO or from certain lattice assumptions.

2.1.6 Tree-based group DH

There is a very large body of literature on tree-based group key agreement schemes. An early example is the "audio teleconference system" of Steer et al. [42], and the seminal academic work is perhaps Wallner, Harder, and Agee [44] or Wong, Gouda, and Lam [46]. Later examples include [6, 9, 13, 23, 25, 27, 30, 47], among many others. Roughly, these protocols assign private DH keys to leaves of a binary tree, defining

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

Table 1: Asymptotic efficiencies and properties of some group messaging solutions as a function of the group size n, both in the setup phase and for each message sent and with respect to the creating or sending user. "Pairwise" denotes sending a group message repeatedly over pairwise channels to each intended recipient. When counting exponentiations the pairwise cost depends on how many people have ratcheted since the sender's last update. In "local storage", "channel" represents the amount of local storage needed for a single pairwise channel, and "key" the amount for a single symmetric key. We provide concrete measurements in Section 7.

		number of exponentiations	number of encryptions	bandwidth	local storage	PCS
sender keys (Signal)	setup	$4n-4$	$n-1$	$n-1$	$n \times$ channel + $n \times$ key	✗
	ongoing	0	1	1		
pairwise Signal	setup	$4n-4$	0	$n-1$	$n \times$ channel	✓
	ongoing	0 to $n-1$	$n-1$	$n-1$		
our solution	setup	$5n-5$	$2n$	$3n$	$2\log n \times$ key	✓
	ongoing	$\log(n)$	1	$\log(n)$		

knowledge of at least one secret leaf key. A similar construction can be done with ternary trees for the three-party Joux protocol.

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

(i) one secret leaf key λ_j, and

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

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

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 1 on page 12, node 5 must know (but cannot compute just from the g^{λ_j}) the public keys corresponding to all $\boxed{\text{boxed}}$ nodes.

Other agents may be chosen by the messaging system to compute and broadcast public keys at intermediate nodes. For example, Kim, Perrig, and Tsudik [27] describe a system where certain agents "sponsor" their subtrees, broadcasting the public keys which they know. 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.

2.2.1 WhatsApp

WhatsApp implements end-to-end encryption for group messaging using the Sender Keys variant of Signal for all groups of size 3+, using the existing support for Signal in pairwise channels. Sender keys are rotated whenever a participant is removed from a group but otherwise are never changed; 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 if her phone is offline, even if another device is connected.

2.2.2 Facebook Messenger Secret Conversations

Secret Conversations on Facebook Messenger similarly use the Sender Keys variant of Signal for all conversations involving 3+ devices [15]. As in the WhatsApp implementation, Sender Keys are only rotated when a device is removed from a conversation, so compromising a sender key will allow future messages from that user to be eavesdropped.

2.2.3 Signal

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

2.2.4 iMessage

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

2.2.5 SafeSlinger

SafeSlinger [16] 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 group members.

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. Of necessity [11], ART must therefore support stateful and iterated key derivations, while SafeSlinger derives a single group key, and therefore does not provide PCS. In particular, using the unbalanced DH key tree of SafeSlinger, while reducing the computational load on the initiator, would cause ART's key updates to take linear (versus logarithmic) time.

Second, SafeSlinger is a synchronous protocol in which all group members must be online concurrently. This allows them to have commitment, verification and secret-sharing rounds, after which the trust requirements upon all participants are the same. ART, on the other hand, is an asynchronous protocol which allows group creation and message sending even with members who are not currently online; in the initial phase during which not all members have participated in the group, ART thus places additional trust requirements on the creator as discussed in Section 3.2.2.

3 Objectives

Security properties for authenticated key exchange (AKE) protocols are extremely well-studied. We now describe our high-level threat model and security goals for group AKE.

3.0.1 Secrecy and Authentication

Our fundamental goal is confidentiality and authenticity of keys: an active network adversary cannot learn keys shared between Alice and Bob.

3.0.2 Post-Compromise Security

Traditional security models do not provide any guarantees *after* the long-term keys of a participant are compromised: it is not considered an attack to learn Bob's identity key and then impersonate him to Alice. Cohn-Gordon, Cremers, and Garratt [11] defined the notion of *post-compromise security* to cover this scenario, showing that it is achievable through the use of persistent protocol state.

We aim explicitly to achieve a form of PCS in our messaging protocols: if the full state of a group member is compromised (long-term and other derived keys) but the group conversation continues without interference and new, uncompromised keys are derived, 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, and hence it does not meet this form of PCS. The PCS property is a major distinguishing feature of modern two-party messaging protocols, and offers significant protection from adversaries with large resources, forcing them to actively interfere in all communications even after they manage to temporarily compromise a device. We therefore believe that this property is important for modern protocols.

3.0.3 Poor randomness

Security models such as extended Canetti-Krawczyk and its generalisations [12, 29] allow for the corruption of random numbers generated by a party if their long-term keys remain secure. However, many widely-used protocols do not achieve this property—for example, neither TLS nor Signal are secure if both sides' ephemeral keys are revealed. We aim for security even if some random numbers are revealed, as long as not all (in the current tree) are.

3.1 Security properties

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

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

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

PCS [11]. 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 particular goal, and one 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 and which should not hold of ART.

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, as with e.g. Facebook's Secret Conversations.

3.2.2 Malicious group members

In the two-party case, traditional security properties are generally of the form "if the peer to a session is honest then P". With n parties, there is an intermediate type of property: "if $m < n$ members of the group are honest then P". For example, Abdalla et al. [1] give a group key exchange protocol which enables subsets of the group to derive their own key, aiming for security in a subset even if another group member is malicious.

Although these properties are useful, we consider them orthogonal to our core research question. Moreover, because we use standard constructions from the (synchronous) literature, we anticipate that extending our 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 creator might be able to secretly add an eavesdropper to a group without revealing their presence to the other (honest) group members. We remark that a malicious insider could also publish received messages regardless of the underlying protocol. As for any other group member, we regard this context as 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 [9]) 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 $x \coloneqq y$ to denote assigning y to the variable x. We write $x \coloneqq_{\$} S$ 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.

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 (including 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 1.

DH groups

We work in a DH group $\mathcal{G}$ (with generator g) for which the decisional DH problem is hard: given a tuple (g^x, g^y, z_b) where $z_0 = g^{xy}$ and $z_1 :=_{\$} \mathcal{G}$, the advantage of any PPT distinguisher in outputting b is negligible. 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 make use of an injection $\iota(\cdot) : \mathcal{G} \to \mathbb{Z}$ mapping group elements to exponents, allowing us to use a group element g^x as the exponent of a new expression $g^{\iota(g^x)}$.

Signatures and MACs

ART uses two explicit authenticators: a signature to authenticate the initial group setup message, and a MAC to authenticate subsequent updates. We use a standard notation: $s = \textsc{Sign}(m, sk)$ 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. Likewise, $\mu = \text{MAC}(m, k)$ denotes a MAC of the message m with the symmetric key k, and $\text{MAC}\textsc{Verify}(m, \mu, k)$ verifies the MAC and returns a boolean.

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, and we refer to the initial stage 0 of Alice's session $(\text{Alice}, 3)$ using the triple $(\text{Alice}, 3, 0)$. Subsequent stages 1, 2 and so on can then be identified by the triples $(\text{Alice}, 3, 1)$, $(\text{Alice}, 3, 2)$, and so on. We will later use these triples as *session identifiers* or sids.

Derived Keys

Our protocol contains various different classes of secret and public key, which we name as follows.

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.

Session state

We use π to represent the protocol's state:

Definition 1 (State)**.** For agent u, session counter i and stage counter t, the *session state* π is a collection of the following variables.

(i) $\pi.u$, the identity u of the current agent
(ii) $\pi.\lambda$, the leaf key of the current stage
(iii) $\pi.T$, the current tree (with ordered nodes) with *public* keys stored at each node
(iv) $\pi.\mathrm{idx}$, the position of the current agent in the group
(v) $\pi.\mathrm{IDs}$, 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
(vi) $\pi.tk$, the tree key of the current stage
(vii) $\pi.\bar{P}$, the copath of the current agent

Where there are multiple distinct session states under consideration, 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. For example, σ.`active` tracks whether a session is in progress, active or rejected. Values in σ are only used for defining 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) $\sigma.sk$, the agent's secret stage key to be used at the current stage
(iv) $\sigma.\mathrm{status}$, 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
(v) $\sigma.sessk$, the key output by the current stage
(vi) $\sigma.\mathrm{HKeys}$, the set of ephemeral keys honestly generated in the current stage
(vii) $\sigma.\ell[i']$, the number of leaf keys received so far from node i' in $\pi.T$ (when $i' = \pi.\mathrm{idx}$, this is the number of leaf keys that (u, i) has generated so far).

Definition 3 (sid)**.** By $sid(\pi, \sigma)$ 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.

5 Design

We build on the tree-based group DH schemes described in Section 2.1.6, with informal explanations of our Asynchronous Ratcheting Tree (ART) algorithms in Sections 5.1 and 5.2, and precise definitions in pseudocode in Section 5.3. An example tree, path and copath appear in Figure 1.

5.1 Asynchronous Ratcheting Tree Construction

As discussed in Section 2.1.6, distributing the public keys on each agent's copath has normally led to a number of interactive rounds in previous tree DH protocols. We show now that these interactive rounds can be avoided, by using asymmetric *prekeys* together with a one-time asymmetric *setup key.*

Prekeys were first introduced by Marlinspike [32] 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

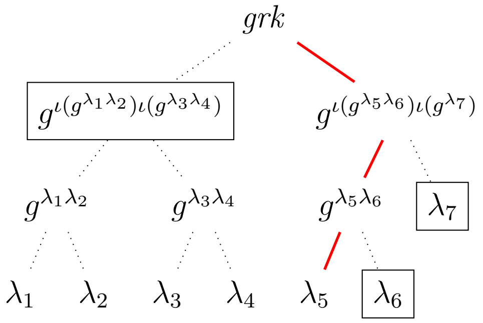


Figure 1: Example computation of a tree key from secret key λ_5. Recall that the λ_j denote leaf keys, and $\iota(\cdot)$ denotes the injection from group elements to integers. The path from λ_5 to the root is marked in solid red, and the boxed nodes lie on its copath. An agent who knows λ_5 and the public keys of the boxed nodes can compute grk.

key infrastructure at the same time as long-term identity keys, act as initial messages for a one-round authenticated key exchange protocol, and allow for handshakes with offline peers.

We introduce in addition a one-time DH *setup key*, generated locally by the creator of a group and used only during session creation. 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 an 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 keys for each peer, she executes a one-round authenticated key exchange protocol to derive leaf keys $\lambda_1, \ldots, \lambda_{n-1}$. Using these generated leaf keys together with her new leaf key λ_0, she builds a DH tree whose root holds the initial group key.

We do not force a particular instantiation of this one-round key exchange protocol. 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 is with a strong authenticated key exchange protocol; in Section 6.3 we discuss this version.

To share the initial group key with her peers, Alice sends

(i) the public prekeys (EK_i) and identities (IK_i) she used to create the group,
(ii) the public setup key SUK,
(iii) the tree T of public keys, and
(iv) a signature of the previous data (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 λ_i using the secret ephemeral key corresponding to their public ephemeral key used by Alice, second they extract their copath of public keys from the tree, and finally they iteratively exponentiate with the public keys on the copath until they reach the final key, which by construction is the tree key tk. They can then compute the stage key sk from tk.

We give a pseudocode definition of these algorithms in Figure 5, Algorithms 1,2,3.

5.2 Asynchronous Ratcheting Tree 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

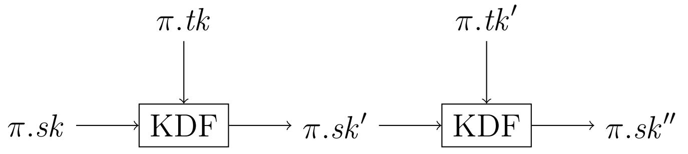


Figure 2: Derivation of stage keys $\pi.sk$. When a new tree key $\pi.tk$ is computed (as the root of a DH tree), it is combined with the current stage key to derive a new stage key $\pi.sk'$, etc. This "chaining" of keys is an important ingredient for achieving PCS. Note that the ART KDF also includes π.IDs and $\pi.T$, per Algorithm 2 on page 15.

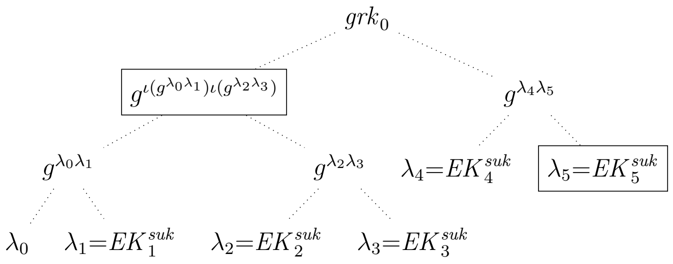


Figure 3: Alice sets up a new tree with herself and five other agents. The copath of Agent 4 is shown boxed.

[11] 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.

An important insight is that 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. Since Alice can compute (ii) in logarithmic time and broadcast it to the group along with her new leaf key, this update is both efficient and asynchronous, allowing for fast, noninteractive key updates.

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 the previous stage key.

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

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

5.2.1 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 [11] 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 2.

5.3 Algorithms

We give pseudocode algorithms for all of the operations in our design in Figure 5. As an example, consider the situation where Alice wishes to create a group with five other agents,

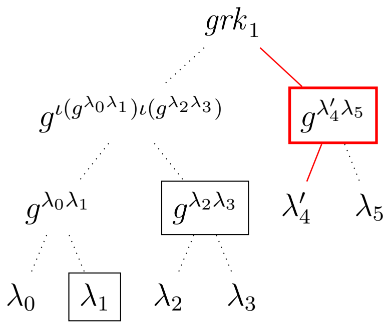


Figure 4: Agent 4 updates their leaf key. The path of Agent 4 is shown in solid red, and the copath of Alice (Agent 0) is shown boxed.

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 3.

She then sends each agent their respective copath and the prekey she used to set them up in the tree, along with the identities of the other group members and the public setup key. For example, the agent Edward at 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 4. 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 group key exchange protocols, and instantiate it with an *unauthenticated* version of our construction in which the initial leaf keys are derived directly from the setup key and prekeys. This allows us to capture the core security properties of the key updates, including 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 a game-hopping argument.

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 [34] (extended with the static-static DH key to provide more resilience against bad randomness and the KCI attack described in [28]), and verify its authentication property using the Tamarin prover, modelling the tree construction abstractly as a "black-box" derivation from the leaf keys.

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
    $\pi.\lambda \overset{\$}{:=} \mathrm{DHKeyGen}()$
    $suk \overset{\$}{:=} \mathrm{KeyExchangeKeyGen}()$
    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 $(\lambda_i, sid(\pi, \sigma))$ to $\sigma$.HKeys
    $T_{\text{secret}} := \textsc{CreateTree}(\lambda_0, \lambda_1, \ldots, \lambda_{n_{\text{peers}}})$
    $\pi.T := \textsc{PublicKeys}(T_{\text{secret}})$
    $\pi.\text{IDs} := \pi.IK, IK_1, \ldots, IK_{n_{\text{peers}}}$
    $\pi.\text{EKs} := \pi.EK, EK_1, \ldots, EK_{n_{\text{peers}}}$
    $x := \pi.\text{IDs}, \pi.\text{EKs}, SUK, \pi.T$
    $m := (x, \textsc{Sign}(x; \pi.ik))$
    $\pi.tk := (T_{\text{secret}})_{0,0}$
    $\pi.\text{idx} := 0$
    $\sigma.\ell[\pi.\text{idx}] := 1$
    $\sigma.\ell[x] := 0$ 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 (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
    $k := \iota\big(LK^{rk}\big)$
    return $(\text{node}((L, lk), (R, rk)), k)$

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

function Copath($T, i$) // where $i$ is the index of the leaf and $|T| = \#$leaves
    if $i <$ LeftSubtreeSize($|T|$) then // i is in the complete left subtree
        return $g^{T_{1,1}}$, Copath($T_{1,0}, i$)
    else// i is in the possibly incomplete right subtree
        return $g^{T_{1,0}}$, 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
    $\pi.sk := \mathrm{KDF}(\pi.sk, \pi.tk, \pi.\text{IDs}, \pi.T)$
    return
```

Algorithm 3 Receiving a setup message as agent at index i

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

Algorithm 4 Agent updating their key

```
procedure UpdateKey
    $\pi.\lambda \overset{\$}{:=} \mathrm{DHKeyGen}()$
    $\sigma.\ell[\pi.\text{idx}] := \sigma.\ell[\pi.\text{idx}] + 1$
    add $(\pi.\lambda, sid(\pi, \sigma))$ to $\sigma$.HKeys
    $nks := \textsc{PathNodeKeys}(\lambda, \pi.\bar{P})$
    $x := \pi.\text{idx}, NKS_1, \ldots, NKS_{|\bar{P}|}$
    $\pi.tk := nks_0$
    $m := x, \textsc{Mac}(x; \pi.sk)$
    $\sigma.t := \sigma.t + 1$
    DeriveStageKey()
    return $m$
```

Algorithm 5 Processing another agent's key update

```
procedure ProcessUpdateMessage($m$)
    $x, \mu := m$
    assert MACVerify($x, \mu, \pi.sk$)
    $j$, nks $:= x$
    $\nu := \textsc{IndexToUpdate}(\lceil \log_2 n \rceil, 0, \pi.\text{idx}, j)$
    $\pi.\bar{P}_\nu := U_\nu$ // 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 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 IndexToUpdate($h - 1, n + 1, i - 2^{h-1}, j - 2^{h-1}$)
    return $n$ // otherwise return index where they differ
```

Figure 5: Pseudocode descriptions of the algorithms in our ART design. Informal explanations can be found in Section 5. We use **procedure** to denote subroutines which are used by the protocol algorithms PSend and PRecv, and **function** to denote ones which are not. Procedures operate on and mutate the agent's current state π and σ, receive an optional input message and return an optional output message, which are received from and returned to the adversary. 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.

6.1 Computational Model

We build on the multi-stage definition of Fischlin and Günther [17], 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 is given a set of queries through which it can interact with the challenger, including the ability to relay or modify messages but also to compromise certain secrets. The adversary eventually chooses a so-called Test session and stage, receiving—uniformly at random—either its true key or a random key sampled from the same distribution. It must then decide which it has received, winning the game if the guess is correct. Thus, a protocol which is secure in this model enjoys the property that an adversary cannot tell if the true keys are replaced with random values.

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 $\mathcal{A}$ 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 $\mathcal{A}$).

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) $\text{Activate}(x, \rho, \mathit{peers}) \rightarrow \pi$: 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) $\text{PRecv}(\pi, m) \rightarrow \pi'$: an agent receives a message m, updating their protocol state from π to π'

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

Definition 5 (Adversary queries)**.** We allow the adversary access to the queries defined in Table 2.

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

6.2 Unauthenticated Computational Analysis

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

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

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 $\text{PRecv}(\pi, m)$ as follows. For a session with $\sigma.t = 0$, validate that m is of the expected format for $\textsc{ProcessSetupMessage}$, and if so then execute $\textsc{ProcessSetupMessage}$

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 $v_1, \dots, v_{n-1}$ ($n \le \gamma$), 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.
$\mathsf{ARecv}(u, i, \mathrm{data})$	Given a session (u, i) with state π, execute $\pi', m := \mathrm{PSend}(\pi, \mathrm{data})$, 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 $\pi.\mathit{sessk}$ 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.

and return the result. For a session with $\sigma.t > 0$, validate that m is of the expected format for ProcessUpdateMessage, and if so then execute ProcessUpdateMessage, returning the result.

We define $\mathrm{PSend}(\pi, \mathrm{data})$ as follows. Validate that data is one of "create-group" or "update-key", or else abort, setting the session state to `reject`. Then, if data is "create-group", execute SetupGroup and return the result; if data is "update-key", execute UpdateKey and return the result.

Definition 6 (Matching)**.** We say that two stages with respective sids (u, i, t) and (v, j, s) *match* if they both have $\sigma.\mathrm{status} = \texttt{accept}$ and moreover have derived the same key.

Definition 7 (Freshness of a copath)**.** Let $\bar{P} = \bar{P}_0, \dots, \bar{P}_{|\bar{P}|-1}$ be a list of group elements representing a copath and let $\Lambda = \lambda_0 \dots \lambda_{n-1}$ 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 some stage whose $\mathit{sid}(\pi, \sigma) = (u, i, t)$ such that $(\lambda_j, \mathit{sid}(\pi, \sigma)) \in \sigma.\mathrm{HKeys}$, and

  (b) no $\mathsf{RevRandom}(u, i, t)$ 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 8 (Freshness of a stage)**.** We say that a stage with sid (u, i, t) deriving key *sessk* is *fresh* if

(i) it has status `accept`,
(ii) the adversary has not issued a $\mathsf{RevSessKey}(u, i, t)$ query,
(iii) there does not exist a stage with sid (v, j, s) such that the adversary has issued a query $\mathsf{RevSessKey}(v, j, s)$ whose return value is *sessk*, and
(iv) one of the following criteria holds:

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

Intuitively, a stage is fresh if *either* all of the leaves in the current tree are 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 1 (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. Indeed, for the creator's first stage to be fresh we must have that their first copath is fresh; since their first copath depends on all the generated λ_j, which were all added to σ.HKeys during the creator's first stage, the adversary is not permitted to issue a $\mathsf{RevRandom}$ query against that stage until all the λ_j from the initial stage have been revealed. This captures the additional requirements discussed in Section 3.2.2.

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. Specifically:

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.

PFS is captured through clause (iv)b 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)a. 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)a stages $j + 1, j + 2, \ldots, t$ are fresh as well. Our model thus requires their keys to be indistinguishable from random to such an adversary.

Definition 9 (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

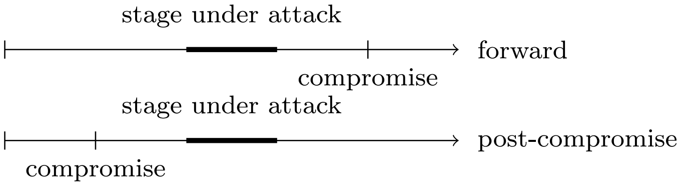


Figure 6: Attack scenarios of forward secrecy and PCS, with the session and stage under attack marked in **bold**. Forward secrecy protects stages against later compromise; PCS protects stages against earlier compromise.

instead. The adversary is allowed to continue asking queries. Eventually the adversary must guess the bit b with a $\mathsf{Guess}(b')$ query before terminating. If the Tested (u, i, t) satisfies fresh and the guess is correct ($b = b'$), the adversary wins the game. Otherwise, the adversary loses.

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

Theorem 6.1. *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 decisional DH assumption, where ι is instantiated as a random oracle, the success probability of any ppt adversary against the key indistinguishability game of our protocol is bounded above by*

$$\frac{1}{2} + \frac{\binom{\mathsf{n_P n_S n_T}}{2}}{q} + \gamma(\mathsf{n_P n_S n_T}^2)^\gamma \left(\epsilon_{DDH} + 1/q\right) + \mathsf{negl}(\lambda)$$

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

Proof sketch (full proof in Appendix A). 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 under 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 the condition of the freshness predicate which it satisfies: either the current copath is fresh or a previous stage was fresh.

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

6.3 Adding Authentication

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

We use 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 a secret key corresponding to a public key in the set.

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 roughly 15 minutes on a modern desktop.

7 ART Implementation

We implemented the ART protocol described in Section 5 in Java, with source code availabele at the URL [36], aiming to show that ART is practical to use for groups of a realistic size, and that it improves on pairwise DH ratcheting (the only other design we know of providing asynchronous PCS) for important metrics. We provide details of our implementation and our timing data in Appendix B.

As ART only provides a single asynchronous ratchet (as opposed to a full Double Ratchet), we also compare it against DH ratcheting with pairwise connections. We note that hash ratcheting could easily be added to ART for a full comparison against the Double Ratchet. We do not benchmark against sender keys, because they do not achieve[1] ART's design goal of PCS. Our

[1] While frequently-rotated sender keys may achieve PCS, the overhead of transmitting the new sender keys converges to that of pairwise channels as the rotation frequency increases.

test implementation ratchets ART on every message but only advances the pairwise DH ratchet for recipients who have not responded to the previous message. This mirrors the behaviour of pairwise Signal without the hash ratchet.

Our test includes encrypting 100 messages between participants in groups of varying sizes, using the two ratcheting solutions to derive encryption keys and encrypt test messages. That is, we use the two algorithms to derive secure channels and measure the performance of these channels. All data are from a 2016 MacBook Pro with a 3.3 GHz Intel Core i7 processor and 16 GB of RAM.

7.1 Metrics

We measure wall-clock time and network bandwidth consumption in various scenarios, but our key 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. We can also separate measurements for the initiator and other parties.

Our asynchronous setup requires the initiator to construct their entire key tree locally, in order to generate the public tree values to send to other group members. Thus, for the group creator it takes linear time in the size of the group. Although this overhead is minimal for the group sizes normally seen in messaging applications, for large-scale use cases we remark that this is a significant performance requirement. We might expect large-scale use cases to be more tolerant of a high setup cost, or perhaps as future work to build up to a large size by dynamically adding group members. We also note that our setup costs for the creator are not significantly higher than the pairwise approach when its channel setup is taken into account, and our total cost over all group members is much less.

7.2 Evaluation

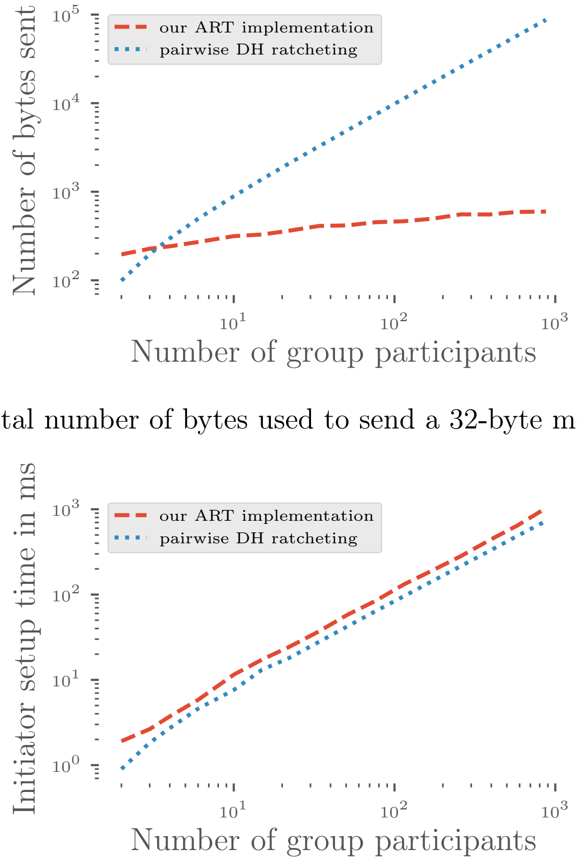
Our results demonstrate that ART is practical for reasonably-sized groups, with key tree setup and message sending both taking a few milliseconds for groups of size ten and on the order of one second for groups of size 1000. ART's performance compares favourably to that of pairwise DH ratcheting, in particular demonstrating a significant advantage for message sending due to its server-side fanout: senders need only encrypt messages once to broadcast them to all group members.

Message sending The advantage is visible particularly clearly in Figure 7a, measuring the total outgoing bandwidth for a sender to send a message (note that the computation cost is directly proportional): for all but the smallest groups, the repeated message encryptions needed for pairwise channels cause significantly more overhead than the logarithmically-sized copaths needed for ART.

The high-level observation is that ART allows for broadcasting the same logarithmic quantity of data to all peers. In contrast, pairwise channels require computing and sending constant-sized data that is different for each peer.

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, due to the fanout: to send a single ART message, each recipient receives a logarithmic quantity of data, while over pairwise channels they receive only the constant-sized message. 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. Additionally, the computational effort needed for encryptions is also significantly lower for ongoing sends in ART.

Setup 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 note that if pairwise channels already exist—for example, a secure messaging application rolling out group messaging—the pairwise design can reuse these existing channels while ART always requires its linear setup time.



(a) Total number of bytes used 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.

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 [44] pointed out the issue of "sender-specific authentication": in a system which derives a shared group key known to all members, there is no cryptographic proof of *which* group member sent a particular message. Various works have discussed such proofs; the most common design is to assign to each group member a signature key with which they sign all their messages. We remark that it is easy to extend our design with such a system; in particular, 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. [23] for a summary of previous work on dynamic groups. In general, since we build on tree-based ideas, our design can support join and leave operations using standard techniques.

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

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

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

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

Chain keys The Signal protocol introduced the concept of *chain keys* to support out-of-order message receipt as well as a fine-grained form of forward secrecy. Instead of using a shared secret to encrypt messages directly, Signal derives a new encryption key for each message from a hash chain. The shared secret derived by our group key exchange can be directly used in the same way, for the same benefits.

9 Conclusion

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

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

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

Acknowledgements The authors would like to thank Richard Barnes for pointing out an error in a previous version of the algorithm.

References

[1] Michel Abdalla, Céline Chevalier, Mark Manulis, and David Pointcheval. "Flexible Group Key Exchange with On-demand Computation of Subgroup Keys". In: *AFRICACRYPT 10*. Ed. by Daniel J. Bernstein and Tanja Lange. Vol. 6055. LNCS. Springer, Heidelberg, May 2010, pp. 351–368.

[2] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *PKC 2006*. Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Vol. 3958. LNCS. Springer, Heidelberg, Apr. 2006, pp. 207–228.

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

[4] Dan Boneh and Mark Zhandry. "Multiparty Key Exchange, Efficient Traitor Tracing, and More from Indistinguishability Obfuscation". In: *CRYPTO 2014, Part I*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8616. LNCS. Springer, Heidelberg, Aug. 2014, pp. 480–499. DOI: `10.1007/978-3-662-44371-2_27`.

[5] Nikita Borisov, Ian Goldberg, and Eric Brewer. "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, 2004. DOI: `10.1145/1029179.1029200`.

[6] Timo Brecher, Emmanuel Bresson, and Mark Manulis. "Fully Robust Tree-Diffie-Hellman Group Key Exchange". In: *CANS 09*. Ed. by Juan A. Garay, Atsuko Miyaji, and Akira Otsuka. Vol. 5888. LNCS. Springer, Heidelberg, Dec. 2009, pp. 478–497.

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

[8] Christian Cachin and Reto Strobl. "Asynchronous Group Key Exchange with Failures". In: *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*. PODC '04. ACM, 2004, pp. 357–366. DOI: `10.1145/1011767.1011820`.

[9] Yi-Ruei Chen and Wen-Guey Tzeng. "Group key management with efficient rekey mechanism: A Semi-Stateful approach for out-of-Synchronized members". In: *Computer Communications* 98 (2017). DOI: `10.1016/j.comcom.2016.08.001`.

[10] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. *A Formal Security Analysis of the Signal Messaging Protocol*. Cryptology ePrint Archive, Report 2016/1013. `http://eprint.iacr.org/2016/1013`. 2016.

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

[12] Cas J. F. Cremers and Michele Feltz. "Beyond eCK: Perfect Forward Secrecy under Actor Compromise and Ephemeral-Key Reveal". In: *ESORICS 2012*. Ed. by Sara Foresti, Moti Yung, and Fabio Martinelli. Vol. 7459. LNCS. Springer, Heidelberg, Sept. 2012, pp. 734–751.

[13] Yvo Desmedt, Tanja Lange, and Mike Burmester. "Scalable Authenticated Tree Based Group Key Exchange for Ad-Hoc Groups". In: *FC 2007*. Ed. by Sven Dietrich and Rachna Dhamija. Vol. 4886. LNCS. Springer, Heidelberg, Feb. 2007, pp. 104–118.

[14] eQualit.ie. $(N+1)$*SEC*. 2016. URL: `https://learn.equalit.ie/wiki/Np1sec`.

[15] Facebook. *Messenger Secret Conversations (Technical Whitepaper Version 2.0)*. Tech. rep. 2017. URL: `https://fbnewsroomus.files.wordpress.com/2016/07/messenger-secret-conversations-technical-whitepaper.pdf` (visited on 05/2017).

[16] Michael Farb, Yue-Hsun Lin, Tiffany Hyun-Jin Kim, Jonathan McCune, and Adrian Perrig. "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, 2013, pp. 417–428. DOI: `10.1145/2500423.2500428`.

[17] Marc Fischlin and Felix Günther. "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. 2014, pp. 1193–1204.

[18] Ian Goldberg, Berkant Ustaoglu, Matthew Van Gundy, and Hao Chen. "Multi-party off-the-record messaging". In: *ACM CCS 09*. Ed. by Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis. ACM Press, Nov. 2009, pp. 358–368.

[19] Matthew D. Green and Ian Miers. "Forward Secure Asynchronous Messaging from Puncturable Encryption". In: *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2015, pp. 305–320. DOI: `10.1109/SP.2015.26`.

[20] The Guardian. *Contact the Guardian securely*. 2017. URL: `https://gu.com/tip-us-off` (visited on 06/2017).

[21] internet.org. *State of Connectivity 2015*. Annual Report. 2016. URL: `https://fbnewsroomus.files.wordpress.com/2016/02/state-of-connectivity-2015-2016-02-21-final.pdf` (visited on 05/2017).

[22] Antoine Joux. "A One Round Protocol for Tripartite Diffie-Hellman". In: *Journal of Cryptology* 17.4 (Sept. 2004), pp. 263–276.

[23] Yongdae Kim, Adrian Perrig, and Gene Tsudik. "Communication-Efficient Group Key Agreement". In: *Trusted Information: The New Decade Challenge*. Springer US, 2001. DOI: `10.1007/0-306-46998-7_16`.

[24] Yongdae Kim, Adrian Perrig, and Gene Tsudik. "Communication-Efficient Group Key Agreement". In: *International Federation for Information Processing (IFIP SEC)*. Paris, France, June 2001.

[25] Yongdae Kim, Adrian Perrig, and Gene Tsudik. "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, 2000. DOI: `10.1145/352600.352638`.

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

[27] Yongdae Kim, Adrian Perrig, and Gene Tsudik. "Tree-based Group Key Agreement". In: *ACM Trans. Inf. Syst. Secur.* (Feb. 2004). DOI: `10.1145/984334.984337`.

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

[29] Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. "Stronger Security of Authenticated Key Exchange". In: *ProvSec 2007*. Ed. by Willy Susilo, Joseph K. Liu, and Yi Mu. Vol. 4784. LNCS. Springer, Heidelberg, Nov. 2007, pp. 1–16.

[30] Sangwon Lee, Yongdae Kim, Kwangjo Kim, and Dae-Hyun Ryu. "An Efficient Tree-Based Group Key Agreement Using Bilinear Map". In: *ACNS 03*. Ed. by Jianying Zhou, Moti Yung, and Yongfei Han. Vol. 2846. LNCS. Springer, Heidelberg, Oct. 2003, pp. 357–371.

[31] Fermi Ma and Mark Zhandry. *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.

[32] Moxie Marlinspike. *Forward Secrecy for Asynchronous Messages*. Blog. Aug. 2013. URL: `https://whispersystems.org/blog/asynchronous-security/` (visited on 05/2017).

[33] Moxie Marlinspike. *Signal Protocol documentation*. 2016. URL: `https://whispersystems.org/docs/` (visited on 05/2017).

[34] Moxie Marlinspike. *The X3DH Key Agreement Protocol*. Ed. by Trevor Perrin. Nov. 2016. URL: `https://signal.org/docs/specifications/x3dh/x3dh.pdf` (visited on 11/2017).

[35] Ghita Mezzour, Ahren Studer, Michael Farb, Jason Lee, Jonathan McCune, Hsu-Chun Hsiao, and Adrian Perrig. *Ho-Po Key: Leveraging Physical Constraints on Human Motion to Authentically Exchange Information in a Group*. Tech. rep. Carnegie Mellon University, Dec. 2010.

[36] Jon Millican. *ART prototype implementation*. 2018. URL: `https://github.com/facebookresearch/asynchronousratchetingtree`.

[37] Adrian Perrig. "Efficient Collaborative Key Management Protocols for Secure Autonomous Group Communication". In: *Proceedings of International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC)*. July 1999, pp. 192–202.

[38] Adrian Perrig, Dawn Song, and Doug Tygar. "ELK, a New Protocol for Efficient Large-Group Key Distribution". In: *Proceedings of IEEE Symposium on Security and Privacy*. May 2001.

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

[40] Dmitry Skiba. *trevorbernard/curve25519-java*. GitHub repository. Feb. 2008. URL: `https://github.com/trevorbernard/curve25519-java` (visited on 05/2017).

[41] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. *Thrift: Scalable Cross-Language Services Implementation.* Tech. rep. 2007. URL: `https://thrift.apache.org/static/files/thrift-20070401.pdf` (visited on 11/2017).

[42] D. G. Steer, L. Strawczynski, W. Diffie, and M. Wiener. "A Secure Audio Teleconference System". In: *Advances in Cryptology — CRYPTO' 88: Proceedings.* Springer New York, 1990. DOI: `10.1007/0-387-34799-2_37`.

[43] Michael Steiner, Gene Tsudik, and Michael Waidner. "Key Agreement in Dynamic Peer Groups". In: *IEEE Transactions on Parallel and Distributed Systems* 11.8 (Aug. 2000), pp. 769–780.

[44] D. Wallner, E. Harder, and R. Agee. *Key Management for Multicast: Issues and Architectures.* United States, 1999.

[45] WhatsApp. *WhatsApp Encryption Overview.* Tech. rep. 2016. URL: `https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf` (visited on 07/2016).

[46] Chung Kei Wong, Mohamed Gouda, and Simon S. Lam. "Secure Group Communications Using Key Graphs". In: *IEEE/ACM Transactions on Networking* 8.1 (Feb. 2000), pp. 16–30.

[47] Zheng Yang, Chao Liu, Wanping Liu, Daigu Zhang, and Song Luo. "A new strong security model for stateful authenticated group key exchange". In: *International Journal of Information Security* (2017), pp. 1–18. DOI: `10.1007/s10207-017-0373-1`.

A Computational Security Proof

Theorem 6.1. *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 decisional DH assumption, where ι is instantiated as a random oracle, the success probability of any ppt adversary against the key indistinguishability game of our protocol is bounded above by*

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

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

Proof. Security in this sense means that no efficient adversary can break the key indistinguishability game against 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 8) 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 *sessk*, 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 [39]. 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 $(v, j, s).\pi.\mathrm{idx} = l$ and $(v, j, s).\pi.T = (u, i, t).\pi.T$ and $(v, j, s).\sigma.\ell[(v, j, s).\pi.\mathrm{idx}] > 0$ 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 $g^{\lambda_1 \lambda_2}$. 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), $g^{\lambda_1 \lambda_2}$ is replaced with a group element g^z sampled uniformly at random, and all subsequent computations upwards along the path of the tree use g^z instead of $g^{\lambda_1 \lambda_2}$.

This is a game hop based on indistinguishability [39]. 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 decisional Diffie–Hellman assumption. Precisely, we aim to show that if a distinguisher $\mathcal{D}$ could efficiently distinguish between the games, then it could be used to break the DDH assumption. This implies that $\mathsf{Adv}_4 \leq \mathsf{Adv}_3 + \max_{\mathcal{D}} \epsilon_{\mathcal{D}}$, where $\epsilon_{\mathcal{D}}$ is the probability that a PPTM $\mathcal{D}$ correctly distinguishes between Games 3 and 4(i).1.

It remains to bound $\epsilon_{\mathcal{D}}$, which we do with a reduction to decisional DH. Specifically, suppose $\mathcal{D}$ is such a distinguisher. We construct an adversary $\mathcal{A}(\mathcal{D})$ against the DDH game as follows: Given DDH challenge g^x, g^y, g^z and the challenge of determining whether or not $z = xy$, $\mathcal{A}(\mathcal{D})$ simulates the hybrid game as the challenger in a fully honest way except it inserts $g^x = g^{x_1}, g^y = g^{x_2}$ and $g^z = g^{x_1 x_2}$.

Our constructed DDH adversary is given 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 g^z and, using this secret, compute all public DH intermediate keys up the tree that depend on 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 $(v, j, t).\sigma.\ell[(v, j, t).\pi.\mathrm{idx}]$. 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 DDH 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 DDH 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 DDH 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:

$$\mathsf{Adv}_3 \leq \mathsf{Adv}_4 + \epsilon_{\mathrm{DDH}} + 1/q$$

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 DDH 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 DDH. At this point, the tree key is no longer a function of the leaf keys—instead, it depends on the keys at the nodes whose children are leaves, each of which has been replaced by a random value, unknown to the adversary. We iteratively replace DH keys using the DDH assumption, starting along the base of the tree and then working our way up until eventually all DH keys in the tree, including the final group key, are independent of each other. It is trivially impossible for the adversary to do any better than guessing in the final game. Given a group size of n, we never need to do more than $n \leq \gamma$ such game hops due to our tree structure. Thus

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

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. □

B Further Measurement Details

Our prototype implementation is available from [36]. 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 [40] of Curve25519 [2]. 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 [41] library and toolchain to serialise messages, to mimic as closely as possible an actual wire format used for RPC.

C Summary of changes

V2.1, January 18th, 2018

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

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

V1.1, July 17th, 2017

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