Efficient and Secure Group Messaging Encryption

Tyler Weitzman
Department of Computer Science
Department of Mathematics
Stanford University
`tylerw@cs.stanford.edu`

June 4, 2020

1 Introduction

Securely asymmetric encryption schemes like RSA have for many years provided cryptographic security to settings in which two users, Alice and Bob, must communicate without being able to pre-establish a shared symmetric encryption key in advance of the communication. Diffie-Hellman in particular provides a convenient way to establish a symmetric key that can then provide a convenient way to communicate securely over a channel using a scheme such as ElGamal Encryption.

In ElGamal encryption, as done in most public key cryptography encryption schemes, a symmetric key is established and then proceeding communication can occur using a symmetric-key based algorithm such as *AES*.

In the last few years and since the Edward Snowden reveal of the NSA's leaked documents, there has been a stronger and stronger desire to provide additional security attributes to encryption schemes intended for sending and receiving messages. Namely, three important features of modern encryption schemes used by applications such as Open Whisper System's Signal application are the features of encryption authentication, break-in recovery and forward secrecy.

Encryption authentication is a straight-forward feature (already existing in all standard encryption schemes) which will not be covered in depth as a topic in this paper: encryption authentication is a feature of an encryption scheme which ensures that the ciphertext has not been tampered with and that the decrypted message is indeed the one that was sent. Authentication is easily achieved with an ElGamal Encryption scheme by choosing a symmetric encryption algorithm that provides the authentication built-in, for example *AES_GCM* is a variant of *AES* encryption that provides such authentication. In fact, any semantically secure symmetric algorithm can be made into an authenticated encryption scheme by proper use of an authentication function combined with it.

As for break-in recovery and forward secrecy: This paper will include formal definitions for both. For the introduction, it suffices to understand that *break-in recovery* refers to the ability of the encryption scheme to recover post being compromised: So even if all of Alice's data is stolen, she should be able to recover and restore the security of the system (assuming that the adversary no longer has access to her new data). *Forward-secrecy* is a security feature in which discovery of the key data for a cipher from today should not allow the adversary to reveal any information about any information that was encrypted in the past. As we will see, both of these features get introduced at once by a simple concept in the encryption scheme.

While modern applications such as Signal do provide such security features in their *double-ratchet encryption algorithm*, which shall be briefly explained in this paper, there are still serious inefficiencies in the encryption of group communication. Namely, when you consider the situation in which Alice, Bob, Carol (and any group of $n \geq 2$) members wish to communicate securely without letting Eve be able to gleam any information on the transmitted data, the current best practice is pair-wise encryption broadcasting; that is, Alice would encrypt the message separately to Bob's key and to Carol's key, and when Carol responds she would encrypt to Bob and Carol.

In effect, the group is the complete graph K_n and each pair of members must be connected over a secure communication channel in order for each member to be able to broadcast messages to the entire group. This naive scheme leads to a linear $O(n)$ number of encryption operations over the group size n which Alice (or any other member) has to perform to send a message. Moreover, the size of the transmitted cipher data is likewise of size $O(n)$.

A less-naive approach is that Alice's first pair-wise broadcasted message should be ciphertext that decrypts to a shared symmetric key that can then be used henceforce using a symmetric encryption scheme like AES_GCM. However, while indeed these messages would subsequently require only $O(1)$ encryption operations, any addition of a new member, removal of a member, or a simple desire to rotate to a new symmetric key (in case the old might have been compromised) would again require $O(n)$ operations on the group. For groups of large size this can be problematic if the group wishes to perform operations such as an *add*, a *removal*, or an *update*, fairly frequently.

In fact, as will be shown in this paper, it is instead possible to achieve an optimization leading to $O(log(n))$ number of operations for accomplishing the establishment of a new symmetric key for the group.

This optimization is possible by use of a tree. I will define more formally how this works in the sections proceeding, however, I first provide here in this introduction a basic bird's eye view understanding for the reader:

Instead of having one member establish the key as in the example above, each member will instead generate their own secret, which will contribute slightly to the final shared group secret. Such a structure will then allow all group members to establish a symmetric key and update the key or the members of the group with a small number of operations.

2 The Ratchet Step

Before proceeding to a definition of the desired security features and a description of the tree structure that accomplishes the desired optimizations, it is important that the reader understands the idea of a **ratchet step.**

A ratchet step in cryptography describes a step that can only move in one direction: Just as mechanical ratchets used for lifting heavy objects can only take steps in one direction and do not move backwards.

The simplest idea for a ratchet step in cryptography would be the use of a one-way function such as a hash function. Let $H : X \times Y \to \{0, 1, \ldots, 2^n - 1\}$ be a collision-resistant hash function. If H acts like a random oracle, meaning that the images seem to be completely randomly mapped in comparison to the pre-images (i.e an adversary cannot guess what the pre-image is by looking at the image) then we consider the function to be one-way because it is too difficult to find the inverse of the image and go backwards. Even the Diffie-Hellman key exchange demonstrates the mechanics of a ratchet step in that it relies on the fact that it is difficult to take the discrete log of a generator g put to a power, so the function $f(a) = g^a$ is effectively one way (under the appropriate group).

Indeed, the Diffie-Hellman exchange algorithm could rather naively be used to achieve forward secrecy and break-in recovery. Namely, suppose that each time Alice sent Bob an encrypted message, Bob then generated a new public-key pair, replied to Alice with his new public key, and the next message from Alice to him would use the new public key. In this fashion, even if the first 3 messages get compromised, any new messages would still be secured (break-in recovery); and if today's key was compromised, all the old messages were encrypted with old keys that may have been deleted by now (forward secrecy). Each generation of a new public-key pair would be considered a ratchet step.

In practice, the way Signal uses this idea is in combination with the idea of a chain generated from a key-deriving function. A key deriving function is a function which derives a seemingly random function from an input (a simple example would be a collision-resistant hash function like SHA256). The symmetric key continues to go through this function to generate the next one, while then also being combined with new Diffie-Hellman values being generated with each message as described above. In this fashion, by combining both a one-way key chain ratchet step along with a new public/private key-pair information ratchet step, only the parties that have access both to the new Diffie-Hellman values as well as all of the old Diffie-Hellman values and chain of keys have the ability to generate the new symmetric key and decrypt messages.

Due to the use of two separate ratchet steps performed at every message, this protocol is known as The Double Ratchet Algorithm. To read the full protocol go to The Double Ratchet Algorithm, available at https://signal.org/docs/specifications/doubleratchet/. This method of message encryption is currently one of the most popular methods for message encryption in the world, with more than 2 billion users' messages being encrypted using this scheme on WhatsApp's messaging platform, which uses the Signal protocol.

3 Security Features

Below I will define more formally our desired features of forward secrecy and break-in recovery. In general, the best practice for defining security in cryptography is by describing an attack game where a challenger encrypts data and provides it to an adversary $\mathcal{A}$ who wishes to gleam information from the encrypted data. If the adversary $\mathcal{A}$ cannot statistically distinguish encrypted data (and therefore perceives it to be entirely random) then the information is considered to be semantically secure. I will forego providing the definition for standard semantic security described but will provide to the reader here my own definition for forward secrecy and for break-in recovery using the model of an attack game and using the same definition for semantic security within those definitions.

For both games, we will let $\mathcal{E} = (E, D)$ define an encryption scheme providing such features, where E is the encryption algorithm and D is the decryption algorithm.

$$E : \mathcal{M} \times \mathcal{K} \times \mathcal{X} \to \mathcal{C} \times \mathcal{X}$$

E is an encryption function mapping from the message space $\mathcal{M}$, the key space $\mathcal{K}$, and the *state* space $\mathcal{X}$. The state space is an addition on a standard encryption scheme and will allow these features to exist. As described in regards to Signal's Double Ratchet step, this state can include information about the chain of derived keys or the new Diffie-Hellman key information. In the case of a tree structure, the state might represent the current values and nodes of the tree structure. The state could at its simplest be a simple counter of natural numbers $\mathbb{N}$ counting the number of total messages sent between Alice and Bob. In fact, such a counter is usually desired even as part of simple ElGammal encryption in order to prevent *replay attacks* where an adversary can send the same cipher text again and convince Alice or Bob that the message was sent from the other due to the valid signature; a signed counter would confirm for Alice or Bob that this is in fact an old message.

Certain parts of the state may be sent publicly, such as public keys, while other parts of the state are kept in sync privately by each party, such as the secret values in a tree structure. The algorithm D decrypts from the $(\mathcal{C}, \mathcal{K}, \mathcal{X})$ space.

$$D : \mathcal{C} \times \mathcal{K} \times \mathcal{X} \to \mathcal{M} \times \mathcal{X}$$

With this encryption scheme, we can now formally define forward secrecy and break-in recovery.

Definition 1 *Forward secrecy:*

For a given encryption scheme $L = (E, D)$, defined over $(\mathcal{M}, \mathcal{K}, \mathcal{C}, \mathcal{X})$, and a given adversary $\mathcal{A}$, we define an *attack game* that runs as follows to define a scheme that is forward secure.

- The challenger picks a random message $m_1 \in \mathcal{M}$, and an initial key $k \in \mathcal{K}$. The challenger initializes the state $x_1 \in \mathcal{X}$ and then computes the cipher $c_1 \in \mathcal{C}$ and sends c_1 to the adversary $\mathcal{A}$. As part of the encryption output the challenger is also left with a new state $x_2 \in \mathcal{X}$. The challenger may perform this step as many time as the adversary would like with different messages before proceeding to the second step, since this is how the situation would be in real life.

- Next, the challenger picks a new random message m_2, and encrypts it using $c_2 = E(m, k, x_2)$ using the new state generated from step 1. The challenger sends the cipher c_2 to the adversary.

- We will assume that a symmetric encryption algorithm is being used as part of the forward secure scheme. Suppose that the challenger was using AES_GCM256 (without loss of generality this can be any semantically secure symmetric algorithm) as his symmetric encryption scheme, such that $c_2 = AES\_GCM256(m, k') = E(m, k, x_2)$ for some $k' \in \{0,1\}^{256}$, the challenger shall now provide this key k' to the adversary.

At the end of this game, the adversary should easily be able to compute m_2 by possession of the AES 256-bit key k', however, if the scheme is forward secure then the adversary will not be able to distinguish c_1 from any random piece of data, and if now given a random message m' alongside the real message m_1, the adversary should be statistically unable to distinguish which message was encrypted to generate the ciphertext c_1. The probability that the adversary guesses correctly minus the probability the adversary guesses wrong, which is the adversary advantage, should be (negligibly) zero. Let W_1 be the event that the adversary guesses correctly and W_2 be the event that the adversary guesses wrongly.

$$Adversary\ Advantage = |Pr[W_1] - Pr[W_2]|$$

Definition 2 *Break-in recovery:*

Break-in recovery is defined by a near equivalent game. The challenger picks a random message $m_1 \in \mathcal{M}$, and an initial key $k \in \mathcal{K}$. The challenger initializes the state $x_1 \in \mathcal{X}$ and then computes the cipher $c_1 \in \mathcal{C}$ and sends c_1 to the adversary $\mathcal{A}$. As part of the encryption output the challenger is also left with a new state $x_2 \in \mathcal{X}$.

Next, the challenger picks a new random message m_2, and encrypts it using $c_2 = E(m, k, x_2)$ using the new state generated from step 1. The challenger sends the cipher c_2 to the adversary.

This time, the challenger reveals to the adversary the initial key $k \in \mathcal{K}$ and the initial state x_1. However, without x_2, which includes information on a ratchet step performed from the original step, the adversary cannot decrypt c_2, and given a random message $m' \in \mathcal{M}$ alongside m_2 should not be able to guess which one was used to generate c_2.

4 A Simple Symmetric Ratchet

Suppose that we are using symmetric encryption (so a shared key has already been securely established by the parties), then we make it into a symmetric ratchet by changing the key in a one-way fashion each time we take a ratchet step (perhaps every message or every hour). Below I provide a simple way of achieving this using just symmetric keys and no public-key cryptography, simply by using a random collision-resistant hash function.

Let $\mathit{key}[0]$ be in the initial agreed symmetric key. Then we recursively define

$$\mathit{key}[i] = hash(\mathit{key}[i-1])$$

For the *ith* message key used for authenticated encryption (say with AES-GCM). Each encryption operation would also include a random nonce for added security.

Does this achieve forward secrecy? Only if the parties can securely delete *consumed* keys – keys which have already been used for their decryption/encryption operations.

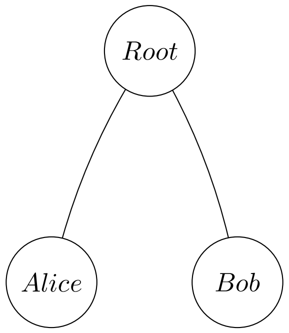
In the forward secrecy Attack Game the key k' given to adversary is equivalently the second key generated in the chain: $\mathit{key}[1] = k'$, if the adversary does not have access to $\mathit{key}[0]$ then he cannot distinguish between the messages without breaking semantic security: In other words, if the adversary had some way of distinguishing between two messages encrypted with $\mathit{key}[0]$ without $\mathit{key}[0]$ then that would break the semantic security of AES, and therefore by proof of contradiction this approach is forward secure. We assume that $\mathit{key}[1]$ does not provide any information about $\mathit{key}[0]$, viewing the hash function under the assumption that it acts like a purely random oracle (even though in reality it is in practice going to be collision resistant and certainly not perfect given that the pre-image space is larger than the range).

5 Using Trees for Key Handshakes

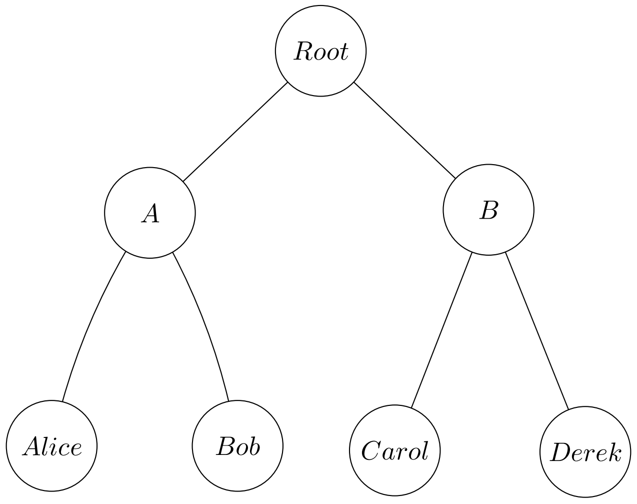
Ratchet Trees are structures using public-key cryptography in order to improve the time complexity efficiency of securing a symmetric key for a group of ends that wants to communicate via an encrypted messaging channel. The ratcheting component in this tree allows new keys to be agreed upon in a group handshake that may occur efficiently on a recurring basis in order to provide break-in recovery (also known as post-compromise security).

Trees are acyclic undirected graphs with one root node that has no parents, numerous intermediate nodes that each have children, and leaves that have no children. In this section we will be discussing left-balanced binary trees, meaning that no more than two children exist per node, and the tree's levels are balanced (no especially deep paths) while maintaining remainder leaves of the tree as left as possible. For the sake of brevity, I will not go into detail on the structure and proofs of tree structure but focus instead on their use in this field of cryptography. I shall assume the reader is familiar with basic tree concepts.

There are numerous methods of using trees to perform the operation of a group handshake. The simplest is the use of a Diffie-Hellman tree. Effectively, the standard Diffie-Hellman exchange algorithm is already itself using a tree, albeit one with only three vertices.



In this figure, we consider each node to store certain values. Namely, for some group G in which discrete log is difficult, in a field generated by the generator g, Alice picks the private key $\alpha \in G$ and publishes the public key g^{α}, and Bob picks the private key $\beta \in G$ and publishes the public key g^{β}. The root's value is the computation available only to Alice and Bob of the shared key $g^{\alpha\beta}$. We can then expand the tree to involve multiple people as follows:



Like before, Alice and Bob both share the symmetric key $g^{\alpha\beta}$, meanwhile, Carol and Derek's private keys γ and δ can be used to generate public keys g^{γ} and g^{δ} and the shared key $g^{\gamma\delta}$. Finally, since both of these keys reduce to elements in G themselves, they can now be used as private keys in a meta-Diffie-Hellman computation of $g^{(g^{\alpha\beta}g^{\gamma\delta})}$, the shared key of the entire tree.

With any such structure, and one that may soon be rotating keys using a ratchet step, we aim to fulfill the following

Definition 3 *Tree invariant*

Each node represents a secret that is only known to its children (and recursively to those children's children). In such a way, the root is by induction always supporting a key known to every group member. Even as node values change this property remains unchanging.

The number of needed operations to initialize this structure (despite the use of a tree) remains $O(n)$. Suppose that there are n group members and therefore n leaves, then just for the first level of computations we must perform $\frac{n}{2} = O(n)$ Diffie-Hellman computations. In fact, so long as the structure is based on each member contributing a secret, there will have to be that many operations to initialize; however, our aim is to allow the tree to be updated to provide break-in recovery without rebuilding the whole tree with linear time complexity.

Suppose that Alice changed her public/private key pair to $(\alpha', g^{\alpha'})$, then rebuilding the tree with the updated value takes fewer operations as half of the tree (either the left of the root or the right root) would not need to be re-computed. However, such an update operation on a Diffie-Hellman tree still remains linear in time-complexity despite the reduced number of operations because half of the tree has to be re-computed and updated if a leaf is changed.

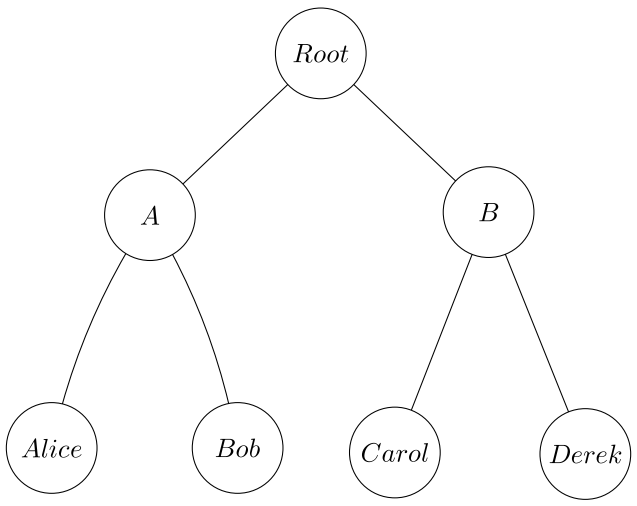
Next, I will review Key Encapsulation Method and its use with trees to achieve a structure in which updating a value can be done with only $O(log(n))$ operations.

6 TreeKEM: Key Encapsulation Method

Key encapsulation refers to the encapsulation of a key inside of a public/private key structure. Let $k \in \mathcal{K}$ be a key in the symmetric key space $\mathcal{K}$ which Alice wants to post to other people in a secure way. Then, using public-key private key cryptography (and any algorithm of choice of a public key system), Alice can transmit this key securely to another member. However, in a group setting, Alice needs to encrypt the key separately for each member. If Alice then wants to use a new encryption key, she needs again to re-encrypt the key separately for each member.

Instead of this inefficient approach, we instead consider a similar approach to a tree with the invariant that each node represents values that are known to all of its children and those children's children.

Let S be a set of members wishing to communicate in a group, and let $S' \subset S$ be a subset of S that may represent a sub-tree in a tree structure as described above. In addition to each individual member having their public/private key pair that might be available in a public authenticated directory, the entire sub-group in this structure will have its own private/private key pair, (s, g^s) where the private key s would be a shared-key between all $|S'|$ members in the group. In this way, new symmetric keys can be encrypted and sent to the entire subgroup with one operation $c = Enc(g^s, k)$



We will consider shortly how such a tree structure can be instantiated in the first place. However, as an example, in the above tree each node would have its own public/private key pair encapsulating a secret symmetric key, where all children and children's children of each node know the private key.

For instance, node A would have its own public/private key pair, where the private key pair is known only to nodes *Alice* and *Bob*, while node B would have its own public/private key pair where the private key is known only to *Carol* and to *Derek*.

Likewise, the root of the tree would have a public/private key pair known to all nodes of the tree. This key pair would be encapsulating a secret value used for the entire group to communicate.

It should be noted that this description above was vastly simplified in order to provide an introductory understanding to the subject. Not any Key Encapsulation Method can be used. While the traditional approach for Key encapsulation is to pick a public/private key pair first and then use that to encrypt a random value or key, it is important for the purposes of this structure (for reasons that will be soon clear) that anybody who has access to the encapsulated key should also be able to decrypt information encrypted for the public key. The simplest way to achieve this property is for the private key to be deterministically generated from the secret value itself.

Definition 4 *Deterministic Key Encapsulation*

Let G be a group with generator g used for public-key cryptography. Let $KEM(x)$ be a key encapsulation method which takes a value $x \in \mathcal{X}$ in some space $\mathcal{X}$ and outputs a private-public key pair of elements in the group G. Then,

$$KEM(x) = (\alpha, \beta, c)$$

Where $\alpha \in G$ is a private key, $\beta \in G$ is a public key, and $c \in \mathcal{Y}$ is optional ciphertext in the ciphertext space $\mathcal{Y}$ containing securely encrypted information about x. Any output of the KEM for a value x in a deterministic KEM would have the same deterministic output value for the private key α every time, such that any holder of x may compute α, and thereby be able to decrypt any information encrypted for β. The ciphertext c need not be used or deterministic, however it makes sense for its value to be an encrypted version of x, say $Enc(\beta, x)$, which would not be deterministic under most secure schemes due to uses of random nonce and salt values.

Key encapsulation methods can achieve this deterministic property by creating a one-way pseudorandom map from the space $\mathcal{X}$ to private-key elements in the group G.

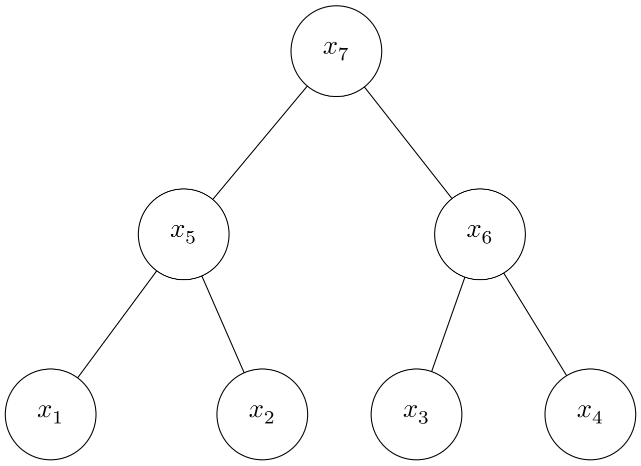
$$H : \mathcal{X} \to G$$

For example, many Elliptic Curve cryptography schemes including Curve25519 and P-256, have a private key space of $\{0,1\}^{256}$, meaning that a standard hash function like $SHA256$ is enough to generate a private key. So an example of a deterministic KEM may be

$$KEM(x) = (SHA256(x), g^{SHA256(x)}, Enc(SHA256(x), x))$$

The encapsulated value x need not be used directly as an encryption key for this to be considered a key encapsulation method. x can represent a random key that is then subsequently used to generate multiple random keys out of it using symmetric key-derivation methods. Indeed, in practice it is most secure to do so, providing x as input key material to produce multiple non-invertible image keys, such that x never needs to be unnecessarily used more than once to encrypt the same thing.

The main advantage of this deterministic property is that we may now build our tree based just on singular random elements in $\mathcal{X}$. Instead of storing Diffie-Hellman values in each node, consider storing random secret values from $\mathcal{X}$ in each node. Consider the following tree for our prior example:



In this tree example, x_1, x_2, x_3, x_4 are generated and known by Alice, Bob, Carol, and Derek respectively.

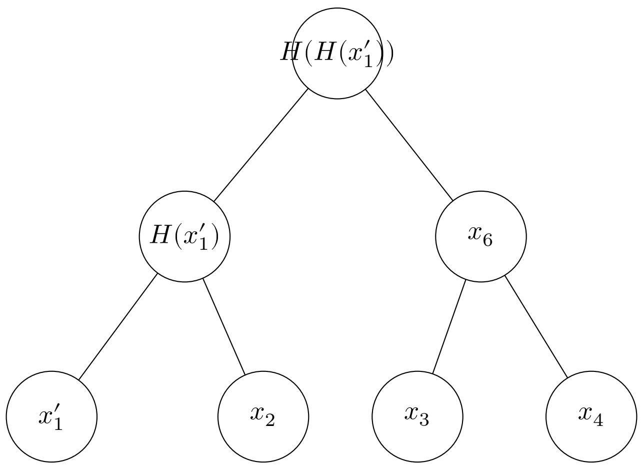
In order to achieve a similar security tree invariant to the one we had in the DH tree, we need the root (and each subroot) value to be computable by its children. For this example, to ensure that the tree invariant holds, it should be possible to generate x_5, x_6, x_7 out of the x_1, x_2, x_3, x_4 values, such that all group members know the group secret x_7.

We can achieve this by defining the following parent child relationship. Let H be a pseudo-random hash or key derivation function. Then, for a non-leaf node $v \in V$ in this tree with vertices and edges (V, E), we define the relationship:

$$val(v \in V) = H(left(v)) \text{ or } H(right(v))$$

Where $left(v)$ is the left child of v such that $(v, left(v)) \in E$ and $right(v)$ is the equivalent right child of v. In the initializing case of leaf nodes $v \in V$, $val(v)$ is randomly generated by the group member associated with that leaf node.

Whether the left or right vertex child will be used depends on the order of operations done in the group. An **update** operation involves a member changing their base key x_i and propagating the change up the direct path of the root, hashing on each level. Suppose for example that Alice wants to update her secret contribution of this tree to x_1'. Then the updated tree would look as follows:



This update operation is how a tree starts in the first place: Each member contributes an initial value to begin with, and each time a member does an update operation it fills out its direct path to the tree. If all members performed this operation there would be a complete tree (though it is possible to get to a complete tree without all doing so).

This tree structure is incredibly powerful because we effectively have a symmetric group key derivable from each binary subtree in this graph. For instance, if $S' = \{Carol, Derek\}$ then it is easy enough to see that they can communicate already by using x_6. More importantly, thanks to the deterministic Key Encapsulation Method, the public key β_6 from the computation $(\alpha_6, \beta_6, c_6) = KEM(x_6)$ can be used by other nodes to encrypt values for this subgroup without being member to S'.

In this fashion, we can now complete the definition for an update operation such that the tree invariant continues holding for the whole tree:

Definition 5 *TreeKEM Update Operation*

1. The updating member computes a new secret value X_0 and propagates it up the direct path to the root, recursively computing $H(X_i) = H(X_{i-1})$ for each subsequent value recursively.
2. For all members that can no longer hash up their direct path to the root, the updating member computes the least number of common ancestors whom if they had the new values would ensure that all group members can reach the root. The algorithm for this computation will be provided later. Once those nodes get the missing value as an encryption to their public KEM key, they can synchronize their view of the tree.
3. The updating member computes the KEM of its newly generated node values and posts publicly the public keys of these nodes to all members of the tree, to be used for future updates.

Members receiving information about updates should verify that the update is authenticated from a verified group member (using a signature scheme), compute similarly which is the common ancestor that they are part of (which S' do they belong to that received the new secret values), use their private key for that subtree to decrypt the value, and then propagate it up to the root using H along its direct path to compute the new key value for the group.

The problem of "which nodes need to receive encrypted values", or as I wrote earlier, computing "the least number of common ancestors whom if they had the new values would ensure that all group members can reach the root" is quite a simple one. Since each node's value is the hash of only one of two children, then the child that was not used to generate the parent is the one that no longer shares knowledge of the root key. It is these *non-updated* children that need to be given knowledge of the parent value via an encrypted ciphertext in order to complete the full tree path to the root node. Therefore, we can identify such nodes simply by looking at the one and only sibling at each level of the direct path. This of course assumes we already have a completed tree. Update operations for trees with blank intermediate nodes are less efficient.

Definition 6 *Algorithm for listing and encrypting updated values*

Let the updating member be vertex $v \in V$ contributing a new secret value x'. Then,

1. Let $sib(v)$ be the sibling of $v \in V$
2. Let β be the public key of the subtree with $sib(v)$ as root, provided by the KEM
3. Compute the value $Enc(\beta, H(x'))$ to be sent to all group members descendant from $sib(v)$ (or to $sib(v)$ if it is a leaf)
4. Assign $v \leftarrow parent(v)$ or end if v has no parent (and is the root)
5. Assign $x' \leftarrow H(x')$
6. Go back to step 1

With these steps, it is clear that we have to encrypt only once per level of the tree (assuming that the tree is complete and not still being filled). If there are $n = |S|$ group members, or leaves in the tree, then the time-complexity for the number of encryption operations is logarithmic in relation, $O(log(n))$, which is our desired out come.

A remove operation also becomes equally simple, essentially being an update that's performed on behalf of a removed member, with a new contribution made on their behalf which they do not know, such that the removed member no longer has access to any current keys in the tree as the entire direct path from their node gets blanked and replaced.

Likewise, an add operation becomes simple to do, similarly equivalent to a new node performing an update, though, since they are not yet part of the group, it would likewise be required for another member to perform the add on their behalf. The member performing the add operation would choose a new random secret value for the new group member, add the new group member to the tree, maintaining a left-balanced tree as much as possible so as to work nicely with the algorithm of the update operation, and then perform an update operation on that node's behalf. The member performing the add operation would then provide the remaining group members with the new $log(n)$ cipher-texts needed for them to come in sync with the tree and would encrypt a similar larger state of the tree for the new node, encrypted for their official public key provided by a trusted authenticated party.

7 Conclusions

Post-compromise security can now be efficiently achieved for the group, since it is feasible with such a small number of encryption operations to update the group key regularly. Not only is it faster to compute new keys, but the size of the handshake for doing so using this update operation is quite small, since there only needs to be a logarithmic number of encrypted cipher-texts provided to each group member upon an update.

If the secret group value x of the root is used to generate a root key of a symmetric ratchet chain, in which keys are hashed for the nth time when encrypting the nth message, and all previous copies of x are thrown out, then we can also achieve forward security.

This paper omits technical details of actually implementing the structure and is not meant to provide a guide for implementing an exact protocol. Rather it should serve as an easy introductory understanding to the concepts used in creating such a protocol. It explains the importance and benefit of a TreeKEM structure for encrypted group messaging and showcases the motivations behind many of the structural decisions made with new protocols implementing such structures, most notably the new Messaging Layer Security (MLS) protocol proposed by IETF's MLS work group. MLS is working to replace the double ratchet Signal protocol in order to be a more efficient new standard and provides technical details for a draft protocol using TreeKEM that is at the time of this writing still being written and peer-reviewed.