BeeKEM and Cryptographic Key Agreement in the Local-First Setting

Derek Yen · 2026-10-06 · English

English transcript. Edited transcript of the presentation and discussion on 6 October 2026. Repetitions and clear recognition errors have been corrected while preserving the order, examples, qualifications and questions. Product status and plans reflect the talk date. Mathematical notation has been checked against the slides. Timestamps refer to the edited video.

Introducing the research



Title and collaborators · Frame 00:00:05

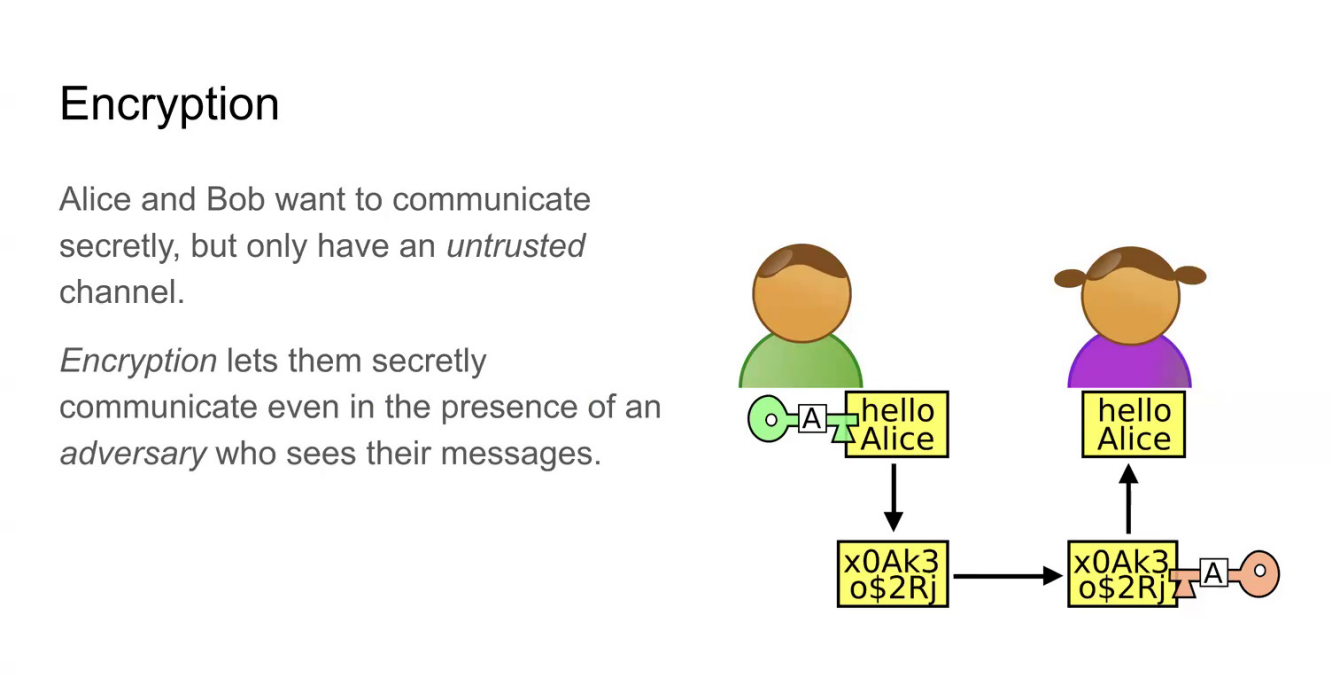
00:00:00 · Derek Yen

Hi, my name is Derek. I'm a third-year PhD student at New York University, and I'm grateful for the opportunity to talk a bit about our work developing BeeKEM, and broadly on cryptographic key agreement in the local-first setting. So this is joint work between academia and industry research at Ink & Switch, with my collaborators Andrés, Liangrun, Martin, John, Sunoo, and Brooke. So, to say a little bit about the structure of the talk.

00:00:29 · Derek Yen

I'll first give a little bit of background on cryptographic key agreement broadly, and then I'll talk about what changes when moving from the centralized setting typically assumed to the decentralized setting, like in local-first. I'll then present a high-level overview of our scheme, BeeKEM, which is a decentralized continuous group key agreement scheme designed specifically with local-first applications in mind, and wrap up with a little bit of a performance evaluation of the implementation of BeeKEM that we have developed.

From encryption to key agreement



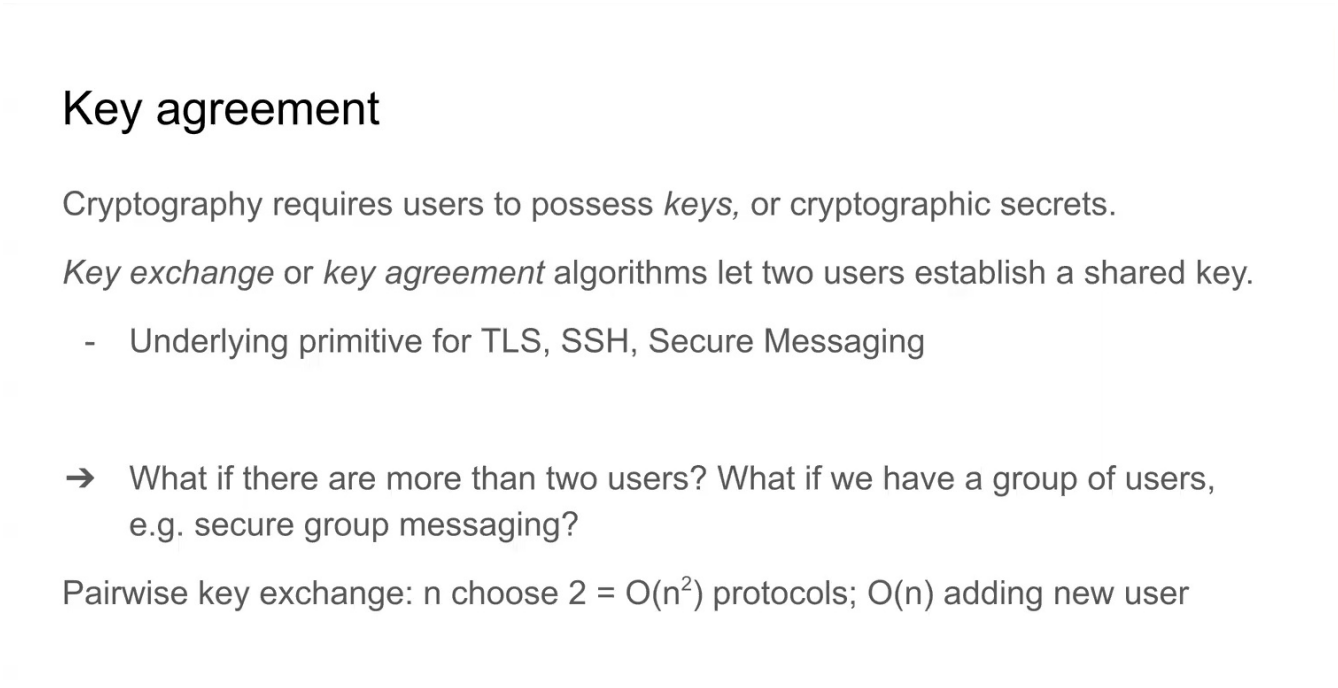
Encryption over an untrusted channel · Frame 00:01:25

00:00:59 · Derek Yen

So, first of all, just starting with the very basics, as I want this to be accessible at all levels of cryptography background. So, what is encryption? Suppose you have the following scenario: that you have two people, Alice and Bob, who want to communicate secretly but only have access to an untrusted channel. Encryption is a type of algorithm which lets them secretly communicate, even in the presence of an adversary who will see their messages.

00:01:28 · Derek Yen

And as this diagram kind of illustrates, a high-level intuition for what encryption does is it takes this message, which is readable, like "Hello, Alice," and then using this secret, which we call a secret key, it is possible to convert it into this text which looks random, but where it is actually possible to take that text, which we call a ciphertext, and to decrypt it back into the original message. So, broadly speaking, cryptographic applications require the users to possess what we call keys, cryptographic secrets, kind of analogous to the password for a computer.



Key agreement and groups · Frame 00:03:15

00:02:06 · Derek Yen

And the way that you are able to have these keys known by the different users of an application is through what we call key exchange, or key agreement algorithms, or protocols, where two users can communicate over a public network and establish a shared key such that even an adversary who sees those packets— or, more precisely, those messages— cannot also obtain that shared key. If you've heard of something like Diffie-Hellman key exchange, that's an example of such a key agreement algorithm.

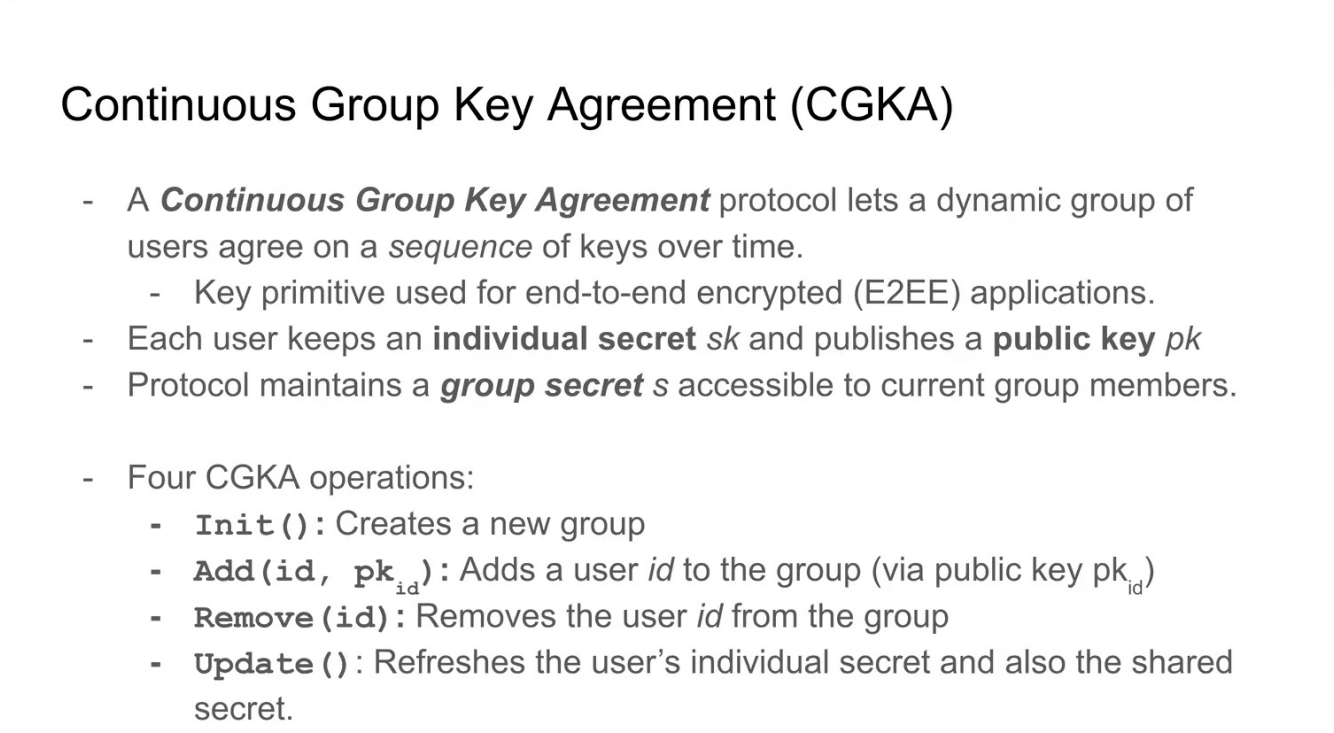
00:02:35 · Derek Yen

This is the underlying primitive used for applications like transport layer security, SSH, and secure messaging applications. So, one very natural situation that might arise is: what if, instead of just two users who want to agree upon a key, what if you have a group of users, such as if you want to have an end-to-end encrypted group chat application? It turns out that you can build this, of course, from pairwise key exchange, by just taking all of the end-to-end pairs of users and having them run a key exchange between themselves, but that requires n-squared calls of this protocol.

00:03:13 · Derek Yen

That's not very efficient if we're thinking about groups that could have n in around hundreds or thousands of users. Furthermore, when you add a new user to the group, they have to exchange a key with every existing member of the group, which is O(n), which is not very feasible. So, a very natural goal then is: what if you could have a better technique for groups, ideally sublinear cost of adding members and doing these types of things?

Continuous group key agreement



CGKA and its four operations · Frame 00:05:25

00:03:39 · Derek Yen

Indeed, this is a line of research that's been very fruitful. This defines a new type of cryptographic primitive, a type of key agreement, called continuous group key agreement. So, as the name suggests, a continuous group key agreement protocol lets a group of users agree on a sequence of keys over time. That's the sense in which we mean continuous. When I say a dynamic group of users, what I mean is that the group does not have to be fixed in the beginning.

00:04:03 · Derek Yen

You are allowed to add new users to the group who should gain access to the secret, and you're able to remove users from the group who should lose access to the secret. And I'll talk a little bit more in a bit about why it's not just one key but a sequence of keys. So, CGKA is the primitive used for end-to-end encrypted applications broadly, so things like messaging but other things beyond that, potentially.

00:04:29 · Derek Yen

In such a scheme, every member of the group will have an individual secret, which I'll denote as sk, and will also publish a public key, pk. The CGKA protocol maintains a group secret, s, which should be accessible only to current group members. More formally, anyone holding an individual secret sk corresponding to a public key pk in the group can access it. There are four basic CGKA operations.

00:04:58 · Derek Yen

There's init, which creates a new group; add, which lets you add a new user to the group with a certain id via their public key, pk id; you can use remove to remove a user from the group; and there's this fourth operation called update, which causes the calling user to refresh their individual secret and their public key, and also causes the shared secret, s, to refresh. And in a little bit I'll explain a bit more about why you would need this, and this ties to why we need a sequence of keys over time.

How operations change the group secret

Example CGKA usage

time	User Operation Log	Group Membership	Group Secret
t = 1	Alice: `Init()`	{Alice}	$\{sk_A\} \rightarrow s_1$
t = 2	Alice: `Add(Bob,`pk_B`)`	{Alice, Bob}	$\{sk_A, sk_B\} \rightarrow s_2$
t = 3	Alice: `Update()`	{Alice, Bob}	$\{sk'_A, sk_B\} \rightarrow s_3$
t = 4	Bob: `Update()`	{Alice, Bob}	$\{sk'_A, sk'_B\} \rightarrow s_4$
t = 5	Bob: `Add(Carol,`pk_C`)`	{Alice, Bob, Carol}	$\{sk'_A, sk'_B, sk_C\} \rightarrow s_5$
t = 6	Alice: `Update()`	{Alice, Bob, Carol}	$\{sk''_A, sk'_B, sk_C\} \rightarrow s_6$
t = 7	Carol: `Remove(Bob)`	{Alice, Carol}	$\{sk''_A, sk_C\} \rightarrow s_7$

Group membership and secrets over time · Frame 00:06:45

00:05:30 · Derek Yen

So, here I just have a kind of example of what CGKA usage would look like. So, on the left here, we have discrete timestamps labeled like t equals 1, 2, and so forth, as well as a log of operations called by different users. On the right, in those two columns, we'll have the set of the current group members, and then also the group secret, as well as which secret keys are able to access the group secret.

00:05:56 · Derek Yen

So, for example, at time 1, t equals 1, Alice creates a new group with init. It initially only contains Alice, and under her secret, ska, it's possible to get the group secret, s1. At time t equals 2, Alice adds Bob to the group, identified with his public key, pkb, for which Bob has a corresponding secret key, skb. The group contains Alice and Bob, and both skA and skB can access the group secret, s2.

00:06:24 · Derek Yen

At time 3, Alice calls update, so this causes her to create a new secret, skA', and also causes the group secret to change. So we see here skA' and skB, Bob's old secret, can still access s3. So, this can continue on, and we notice Bob can update, he can add Carol to the group, and eventually Bob can get removed from the group. And the sequence of keys, or of these group secrets that are generated, are accessible under these different sets of users, correspondingly.

Forward secrecy and post-compromise security

00:06:57 · Derek Yen

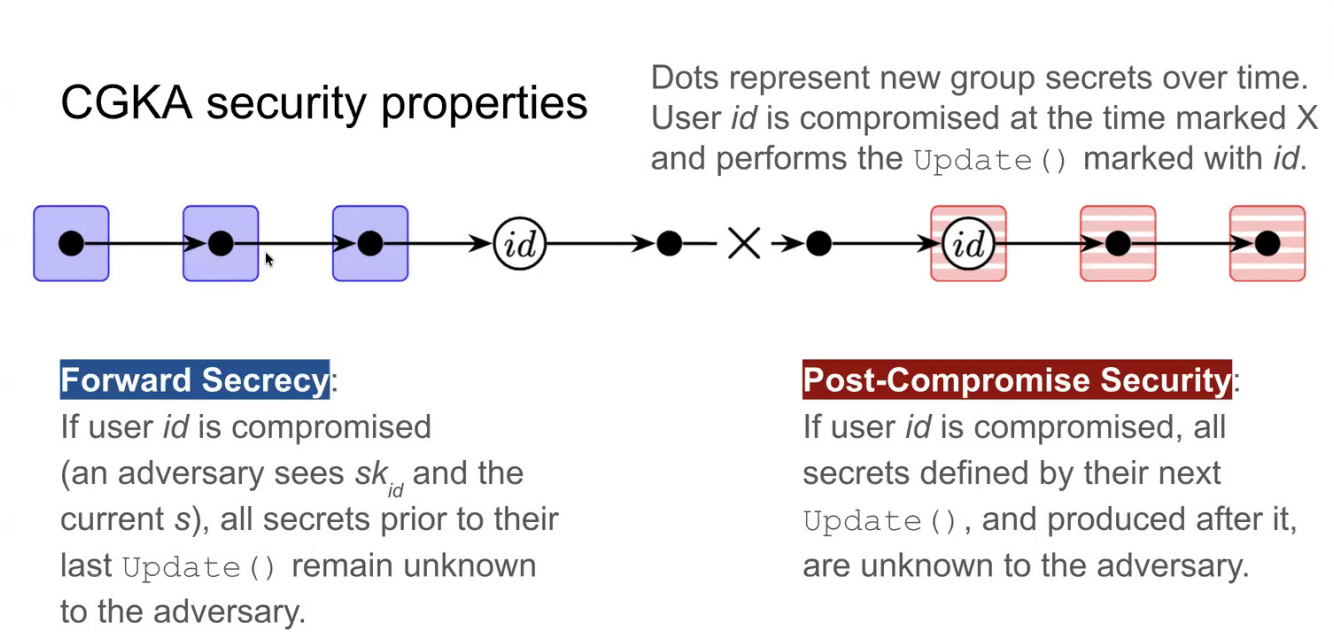
So now I'll talk a little bit about what security properties we expect of CGKA, which relates to this earlier point about defining a sequence of keys over time. So, to kind of sketch an intuition here: because the group is able to change over time, you have to have secrets change over time. For example, when we remove Bob from the group, we have to change the secret from s6 to something else, because if we kept using s6, Bob already has access to s6 under his secret key.

00:07:28 · Derek Yen

So the secret has to change as we perform these operations. Similarly, when we create a new secret, the new secret—or sorry, when we add a new user to the group— they have to be able to access the secret somehow. There's a second effect of changing secrets over time, which is that it also creates a kind of security and robustness against adversaries who might try to break into the group. In particular, suppose we have what we call a snapshot adversary, an adversary who is able to access a user's secret at a given point in time.

00:08:01 · Derek Yen

Well, by having these types of security properties that result from changing the group secret over time, we can have a bounded loss resulting from the adversary breaking in, where losing a secret will only reveal some past messages and not all of them. Looking ahead to the future as well, by rotating secrets over time, the protocol will heal itself. When you lose a secret, the adversary is not able to read all future communication.



Forward secrecy and post-compromise security · Frame 00:09:45

00:08:30 · Derek Yen

So this I will illustrate a bit more concretely with this diagram, and also give you the formal names for those two properties. So these two properties are called forward secrecy and post-compromise security. So in this diagram, which is just showing the evolution of secrets over time, these dots each represent versions of the group secret. The arrows just show that it's changing from time, from left to right being the forward direction in time.

00:08:56 · Derek Yen

And we're going to consider there's a certain user in the group, id, who loses their state at this point marked with the x, as in someone accesses their secret somehow. Formally, that person, that adversary, learns the secret key of id at this point in time, as well as the group secret, s. Furthermore, we assume that these two nodes marked id are updates performed by that user, id specifically, the user who is compromised.

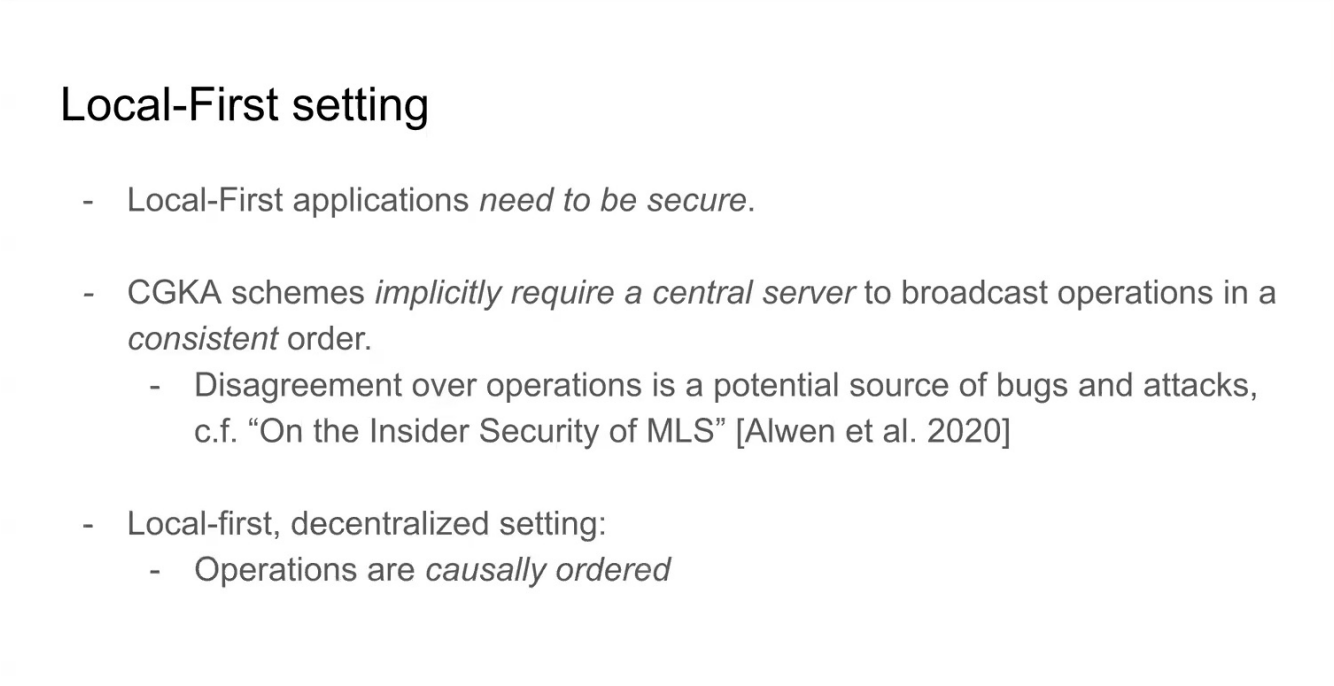
00:09:24 · Derek Yen

So forward secrecy, stated slightly more formally, states that when an adversary compromises user id, as in the adversary sees skid and the current group secret, s, all secrets prior to the last update performed by id should remain unknown to the adversary. So here, with the blue background, I've highlighted group secrets which are protected under forward secrecy. So even though the user id is compromised here at the x, these secrets defined before this last update by id remain protected.

00:09:59 · Derek Yen

Post-compromise security is the future-looking property. When user id is compromised, all secrets defined by their next update and also produced after that update are unknown to the adversary. So here in the red horizontal background are the group secrets that should be protected with post-compromise security. And lastly, I should note here that these non-highlighted group secrets are not protected under these two security properties. Those would potentially be accessible to the adversary. So now I'm going to transition from CGKA, which is implicitly for the centralized setting, into talking about the local-first setting and what changes.

Why end-to-end encryption still needs a server



Causal ordering in local-first · Frame 00:11:35

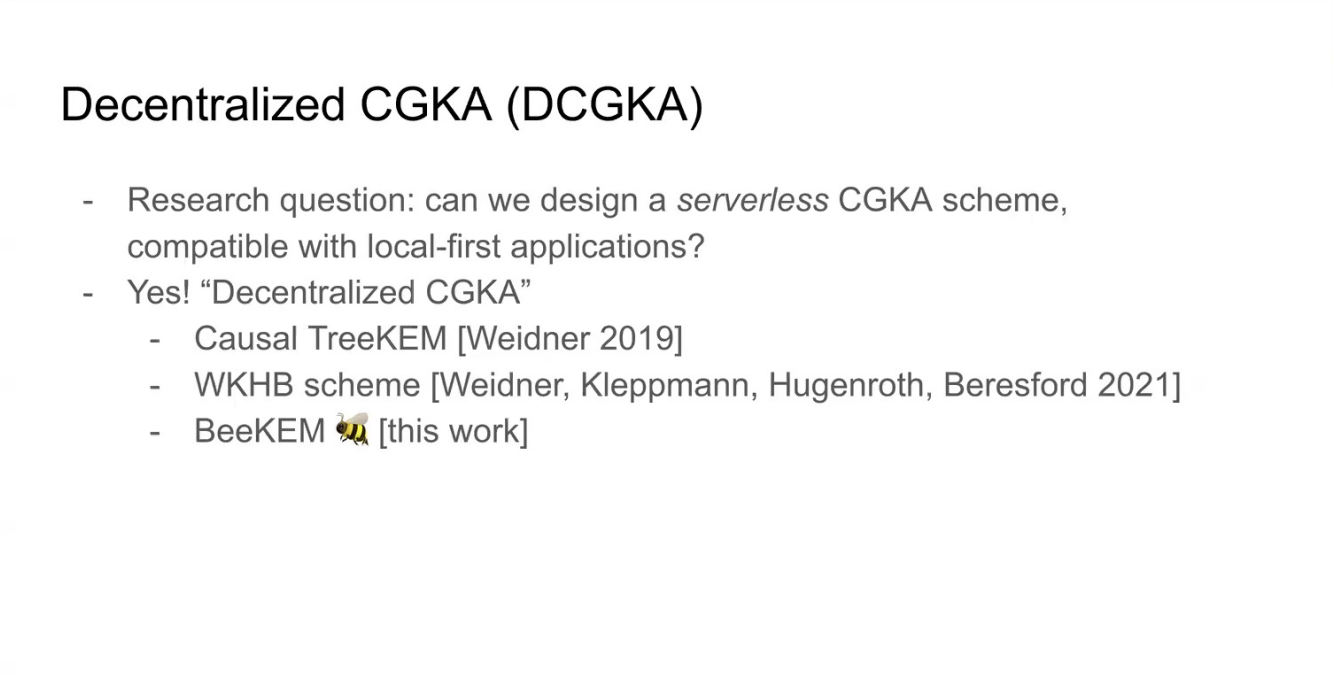
00:10:38 · Derek Yen

So local-first applications, like any others, need to be secure for all the standard reasons of wanting to protect user data. However, there's a bit of a problem, which is that the CGKA schemes, which we have discussed so far, implicitly require a central server to broadcast operations in a consistent order. Even though the ethos of end-to-end encryption is essentially—or sorry, even though the idea of end-to-end encryption is that the central server should not be able to read any traffic, that doesn't mean that the protocol can operate without a central server.

00:11:10 · Derek Yen

The central server has this crucial role in ordering operations. And indeed, if you have situations where there is disagreement over operations, or a scheme which allows users to potentially disagree over operations, this is a source of correctness bugs for your protocol and actually some several viable attacks, which have been studied in prior work. In the local-first decentralized setting, operations are no longer totally ordered but causally ordered, and in particular, the network may suffer partitions and users may go offline for very long periods of time.

From CGKA to decentralized CGKA



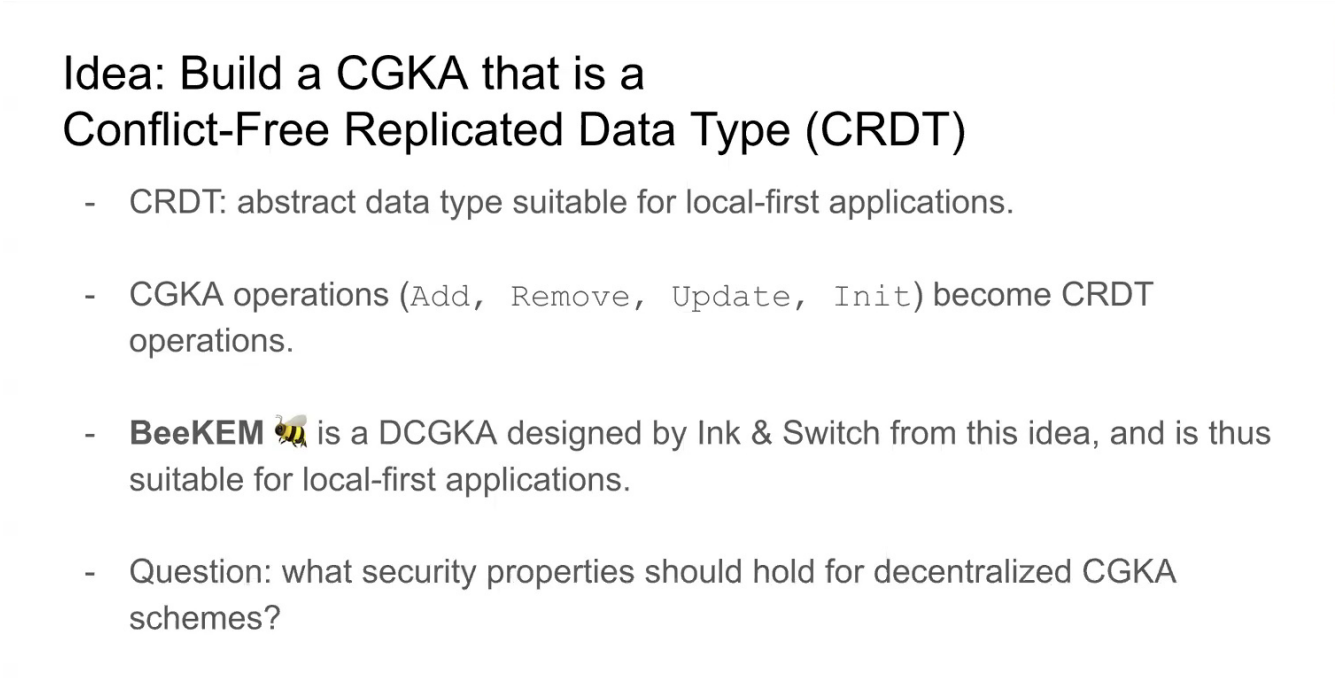
Decentralized group key agreement · Frame 00:12:45

00:11:45 · Derek Yen

So we cannot assume a lot of the nice properties that happen in the centralized setting. So a natural research question is: is it possible to design a serverless kind of CGKA scheme, which would therefore be compatible with local-first applications? And the answer is yes. So this is called decentralized CGKA, or DCGKA. And there's two prior schemes that we know of that developed CGKA for the decentralized setting. There's causal TreeKEM from 2019, and there's the scheme which we've called WKHB, which also formalized the notion of DCGKA, which came out in 2021.

00:12:22 · Derek Yen

Our work, BeeKEM, which we are presenting here, is, to our knowledge, only the third such protocol, though there have been types of protocols in the centralized setting which try to move towards more robustness for reducing dependence on the server in various ways. BeeKEM is the first truly decentral—or I shouldn't say first, but it is the third truly decentralized scheme and the first to achieve optimal asymptotics in certain respects. So the key idea behind BeeKEM is actually to use certain ideas from the design of conflict-free replicated data types, which I will talk about briefly, which have been mentioned in— I have seen some earlier talks in this series— but which I will review a little bit in case people weren't present for that.



Building CGKA as a CRDT · Frame 00:13:55

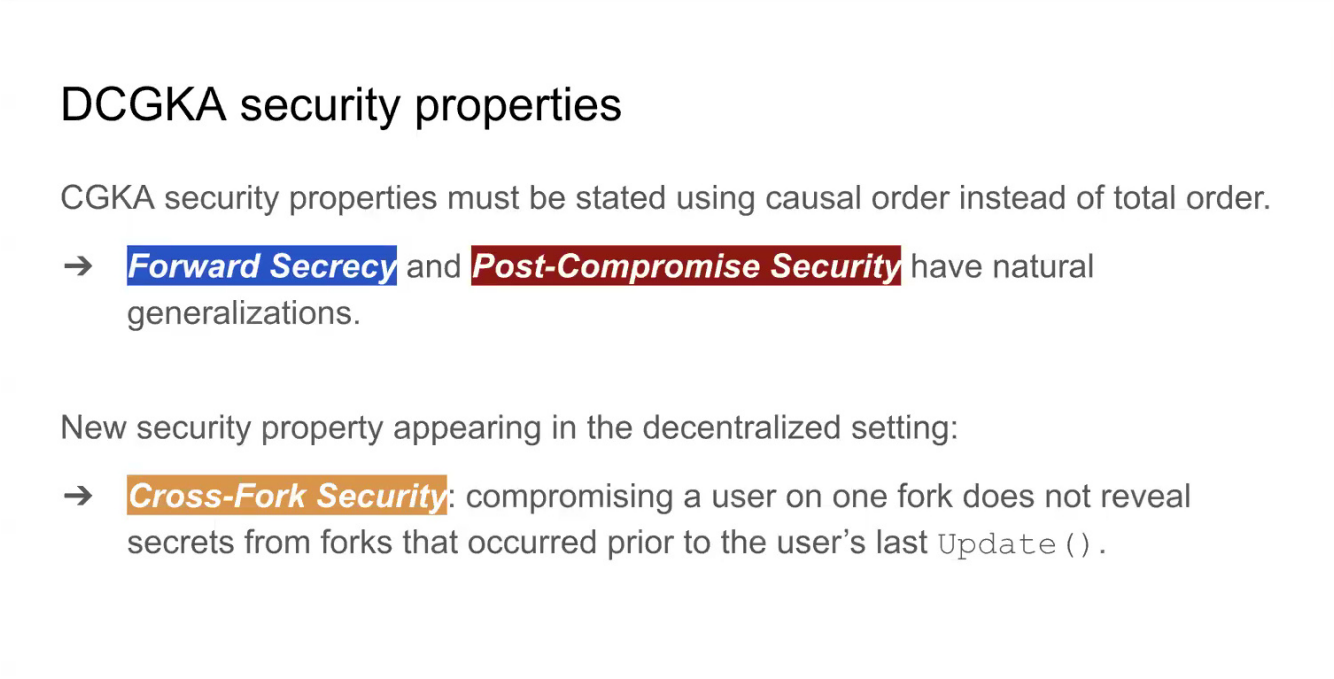
00:13:08 · Derek Yen

A CRDT is a type of abstract data type which is suitable in various ways for local-first applications. It allows you to maintain a data structure among multiple different replicas that individual users can use locally, essentially. And designing this CRDT, we are going to take the CGKA operations and use them as the operations of our CRDT. So these four basic CGKA operations are the ones that we use in our CRDT as well.

00:13:43 · Derek Yen

BeeKEM has been designed precisely from this idea of building a CGKA scheme, which is a CRDT, and has been designed specifically, actually, with local-first in mind. So one natural question when developing a cryptographic protocol is: what security properties should hold for a decentralized CGKA scheme? So we expect that we want the two normal centralized CGKA properties of forward secrecy and post-compromise security. We have to state them slightly differently, because you may recall that they used these ideas of before and after certain updates, and we no longer have a total ordering.

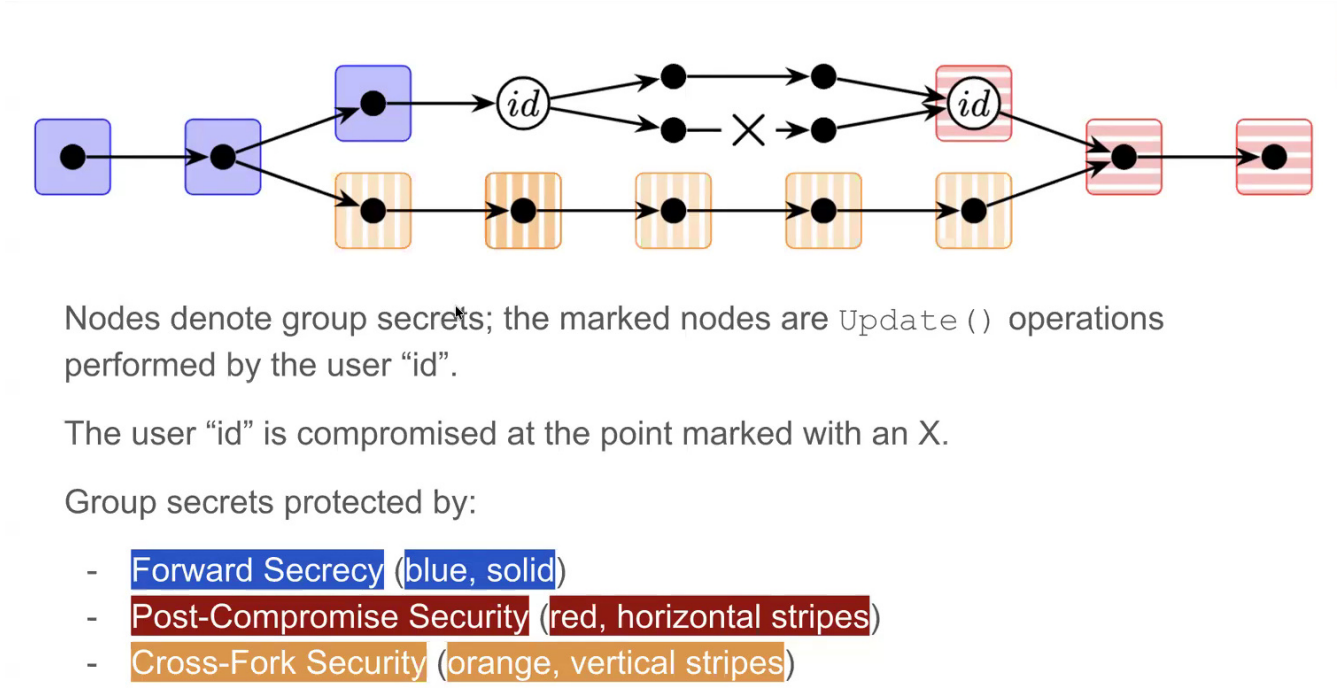
Cross-fork security



Defining cross-fork security · Frame 00:14:50

00:14:20 · Derek Yen

But it turns out that there's a natural generalization of them where you talk about the causal ordering instead of the total ordering. But it is also the case that there is a new security property that appears only in the decentralized setting because of the possibility of concurrency-creating forks. We have a natural security property that applies to those, which we have called cross-fork security. Stated somewhat informally, compromising a user on one fork should not reveal secrets from forks that happened prior to the last user's— to that user's last update.



Which secrets the security properties protect · Frame 00:15:45

00:14:48 · Derek Yen

So to make this a little bit more concrete, I have a diagram again. This is the exact same diagram as before, but we are now viewing this also as an operation graph where nodes may now have multiple successors. Technically speaking, you would encode this as predecessors, but I flipped the arrows as it's a little bit more natural to think about. So in blue, as before—or to back out, as before, these dots represent different versions of the group secret in our DCGKA scheme.

00:15:19 · Derek Yen

We have these two marked dots with id representing a certain user's updates, and that user id loses their state to an adversary at the point marked x. So as before, we have some group secrets which are protected under forward secrecy, marked with the solid boxes in blue, and some that are protected under post-compromise security, marked with the horizontal line red boxes. But we also have some secrets which are protected with cross-fork security, marked with this background that is orange with vertical stripes.

The BeeKEM tree and operation history

00:15:50 · Derek Yen

And as before, there are some group secrets that are not protected under any of these properties. So in particular, we see that this fork which happened after id's last update is not protected under cross-fork security. So now, with this general notion of CGKA established, and now this general notion of DCGKA established, I'm going to talk a little bit at a high level about how the construction of BeeKEM works. So BeeKEM, our scheme, has users maintain two objects on their devices.

00:16:22 · Derek Yen

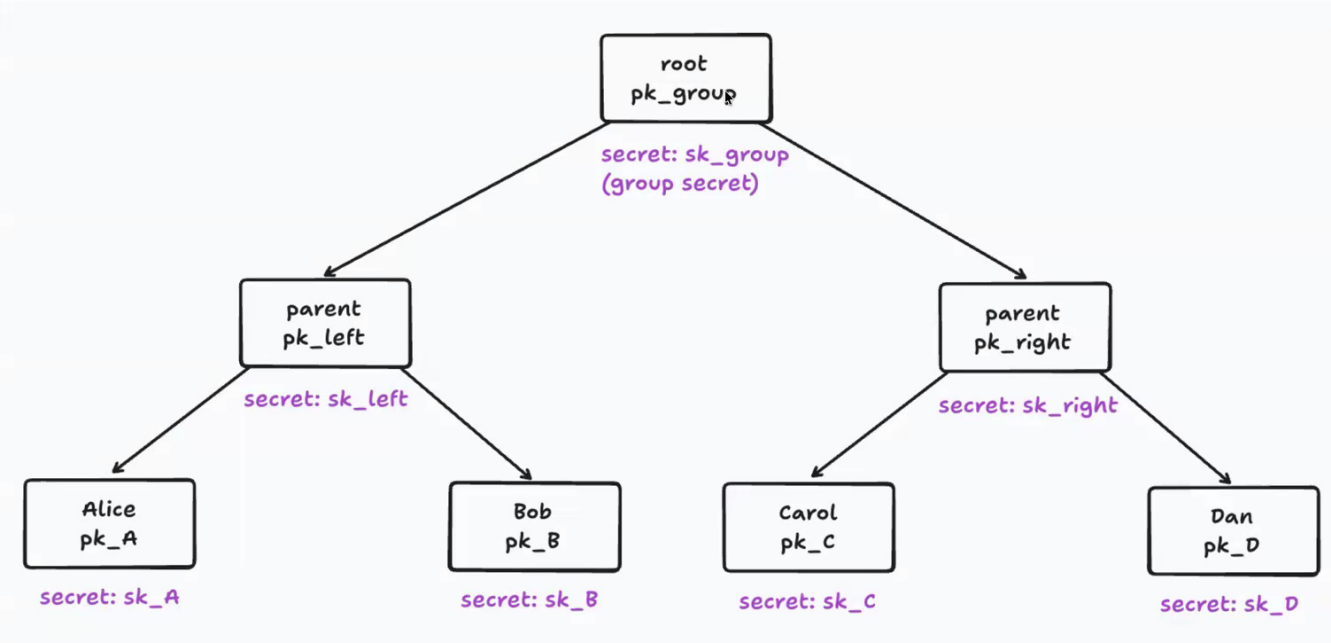
The first is a tree, and the second is the operation history graph. The operation history graph is used in a way analogous to normal CRDTs, where users will record all manipulations of the underlying data structure, and the tree is the actual data structure of interest. The tree has a way of encoding the current members of the group and also a way for users to compute the current group secret. The CGKA operations of interest will modify the tree in various ways, and users will maintain the entire history of operations which they have done themselves and which they have seen.

00:16:59 · Derek Yen

And whenever they perform new operations, they will broadcast them to each other via, for example, a gossip network. So let me tell you a little bit more about the BeeKEM tree, as the operation graph may be a little bit more familiar from prior talks on CRDTs. So with the BeeKEM tree, we will have every group member associated with a leaf of the tree, and the leaf will be labeled with that user's public key, PKA.

00:17:25 · Derek Yen

All of the inner nodes of the tree will have an associated public key and a secret key. And we will aim to maintain this following invariant: if you have a child secret key of any node in the tree, if you have the child secret key, then you are able to get the parent's secret key. And if we succeed in maintaining this invariant, then we immediately have this corollary: all users can access the secret at the root of the tree.



A BeeKEM tree with four members · Frame 00:18:15

00:17:50 · Derek Yen

So that will be what we take as our group secret, because all of the leaves can access the root's secret. So to illustrate this concretely with a diagram, here I have an example of a BeeKEM tree just on four different users. So here, the four leaves are each associated with one of those users: Alice, Bob, Carol, and Dan. So in black, I have all of the information which is published with the tree, and in this purple color are things that are secrets, not necessarily accessible.

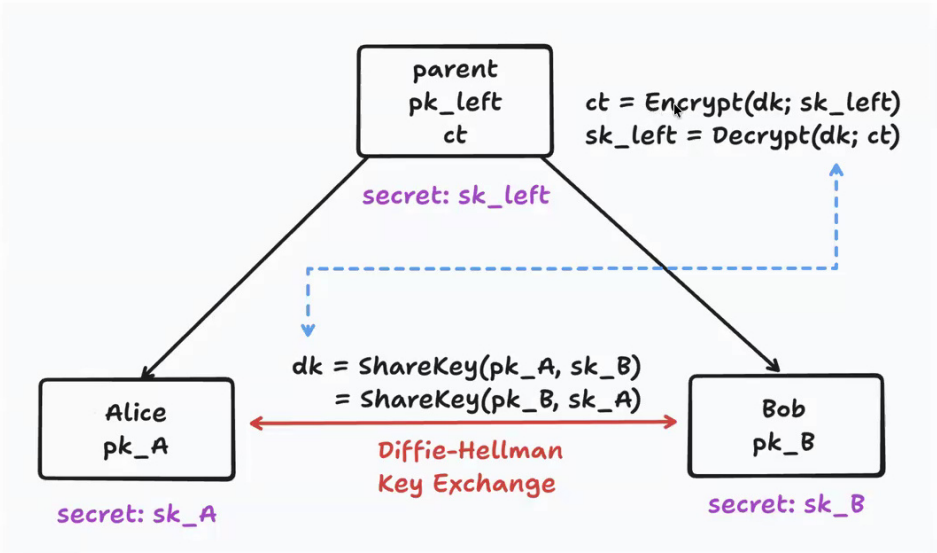
00:18:20 · Derek Yen

So we see that, for example, Carol has at her node just PKC published, but she keeps SKC, her corresponding secret key, to herself. These two inner nodes, PKLeft—sorry, I should say the parents on the left and the parents on the right— have associated public keys, PKLeft and PKRight, and associated secrets. And then here at the root, we have PKGroup and a secret SKGroup, which is the group secret. So let me say a little bit more about how exactly we maintain this invariant, which I've reproduced here at the bottom.

Deriving secrets with Diffie-Hellman

00:18:54 · Derek Yen

So if we have an inner node secret, the inner node's secret key is actually stored encrypted as a ciphertext, encrypted under a different key, but done in a way that it is accessible to either of the children's secrets. Specifically, we will use a two-party key exchange protocol between the two child nodes to obtain a key that is used to then decrypt the parent's secret. I'll show you a diagram in a bit, as there's a few moving parts.



Invariant: if you have a child's secret key, you can get the parent's secret key.
- **Corollary**: all users can access the secret at the tree root.

Deriving a parent secret from child keys · Frame 00:19:45

00:19:20 · Derek Yen

So here I've just reproduced the left subtree from that prior larger tree, and we'll examine just what's happening with one node in the tree, because this is something we'll just recurse up the tree. So let's say that we are Alice, so we have access just to SKA. The way that we have set up the tree: at the parent, there's the parent's public key, but also this ciphertext, CT, which is an encryption of the secret key, SKLeft.

00:19:48 · Derek Yen

And it is encrypted under what I have denoted DK for derived key. So what is this DK? DK is another secret which Alice can access by running a key exchange protocol with her sibling node. So if you haven't seen Diffie-Hellman key exchange before, all you need to know about it in this context is that it allows you to take a public key from someone else and your own secret key, and create a shared secret key that you can access, but also which, crucially, the other party can access symmetrically.

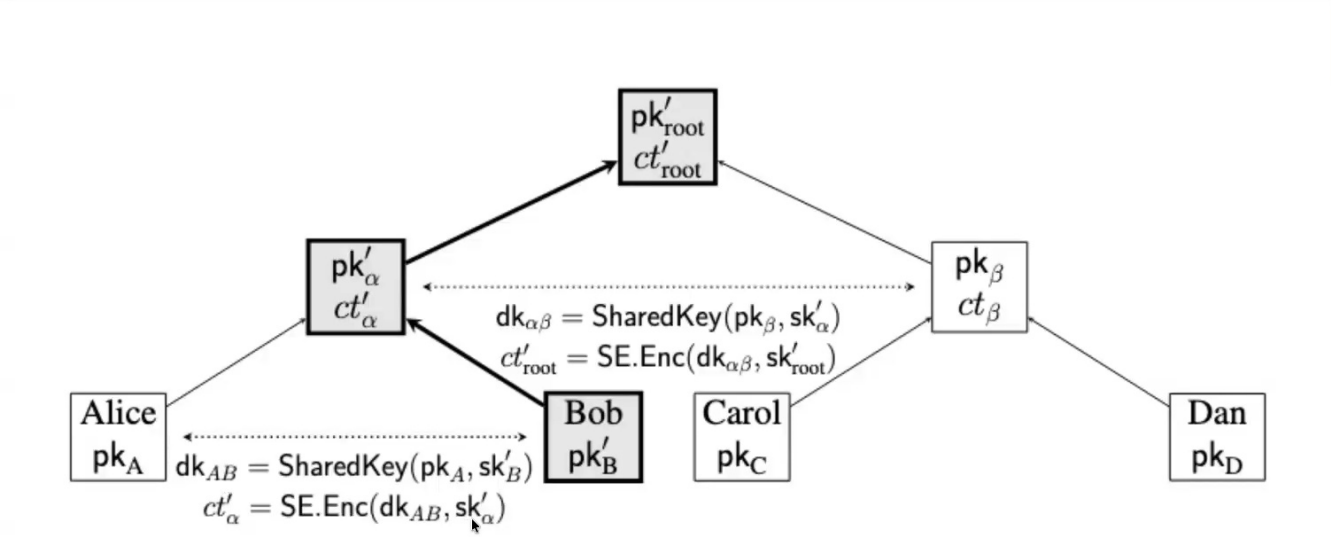
00:20:20 · Derek Yen

Here, dk = ShareKey(pk_A, sk_B) = ShareKey(pk_B, sk_A). Alice is able to compute the second expression because she has sk_A and sees Bob's pk_B. Bob can also access dk using sk_B and Alice's public key, pk_A. This is a special property of certain types of key exchange. BeeKEM specifically uses Diffie-Hellman.

00:20:49 · Derek Yen

So Alice, using SKA, can then get DK. Using DK, she can decrypt this ciphertext and obtain SKLeft, which is the parent's secret. So this structure occurs at all levels of the tree, and thus allows us to establish this invariant that a child secret key lets you get the parent secret key. And then, so concretely, you would use this property and go up the tree to access the secret at the root. So one last thing I'll tell you about is how updates work in this scheme, because one question you might have is also: why do we have this structure?

How an update propagates up the tree



Keys and ciphertexts in Bob's update · Frame 00:22:25

00:21:22 · Derek Yen

Why do we have these ciphertexts that have these secrets? And what happens is, when a user calls the update operation, they will actually sample a new public-private key pair at each level of the tree, starting with their leaf and going up to the root. So in this diagram, I'm showing what happens when Bob performs an update. So the first thing that happens is he will update his node with his new public-private key pair, or he only publishes his public key, PK'B.

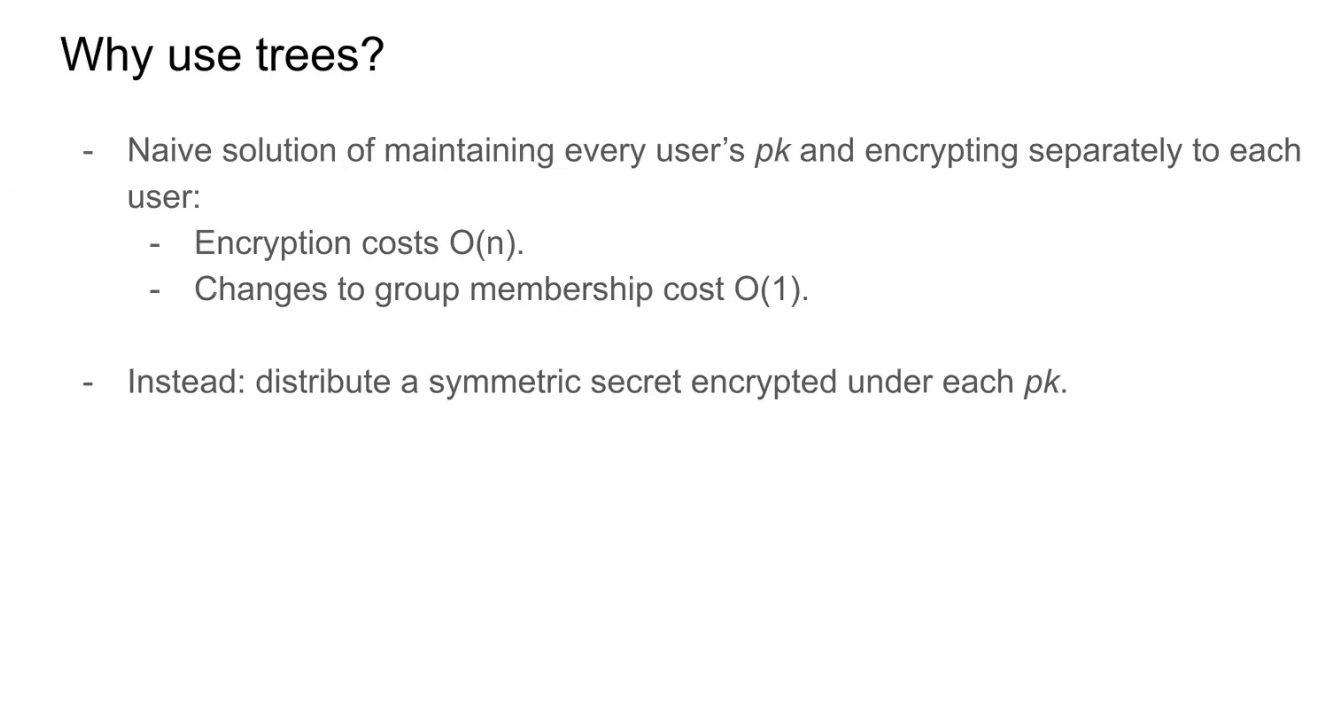
00:21:53 · Derek Yen

Then he looks up the tree and samples a new public-private key pair at his parent, denoted pk_α′. He needs to encode sk_α′ in a way Alice can also access under her secret key sk_A, to maintain the invariant mentioned before. In particular, he uses Diffie-Hellman key exchange, this shared-key algorithm, to compute dk_AB, and uses dk_AB to encrypt sk_α′, creating a new ciphertext at this parent node.

00:22:30 · Derek Yen

He then continues up the tree, repeating this structure. At the next level, he samples a new secret key at the root, the new group secret sk_root′, and encrypts it under the derived key dk_αβ, obtained from sk_α′ and pk_β, the public key of the other inner node. From this structure, it may become clear why we can get logarithmic cost for these operations, although there are a couple of caveats.

Why use a tree?



Encryption and membership costs · Frame 00:23:55

00:23:08 · Derek Yen

So just as kind of a general comment, why is this a tree? Why are we using trees? So think about building a CGKA scheme out of this kind of naive solution where every member of the group publishes a public key, and you encrypt separately to each user. Well, encryption would not be very efficient in this scheme, because it's O(n). You have to encrypt to all n users. But on the other hand, changes to group membership are very efficient.

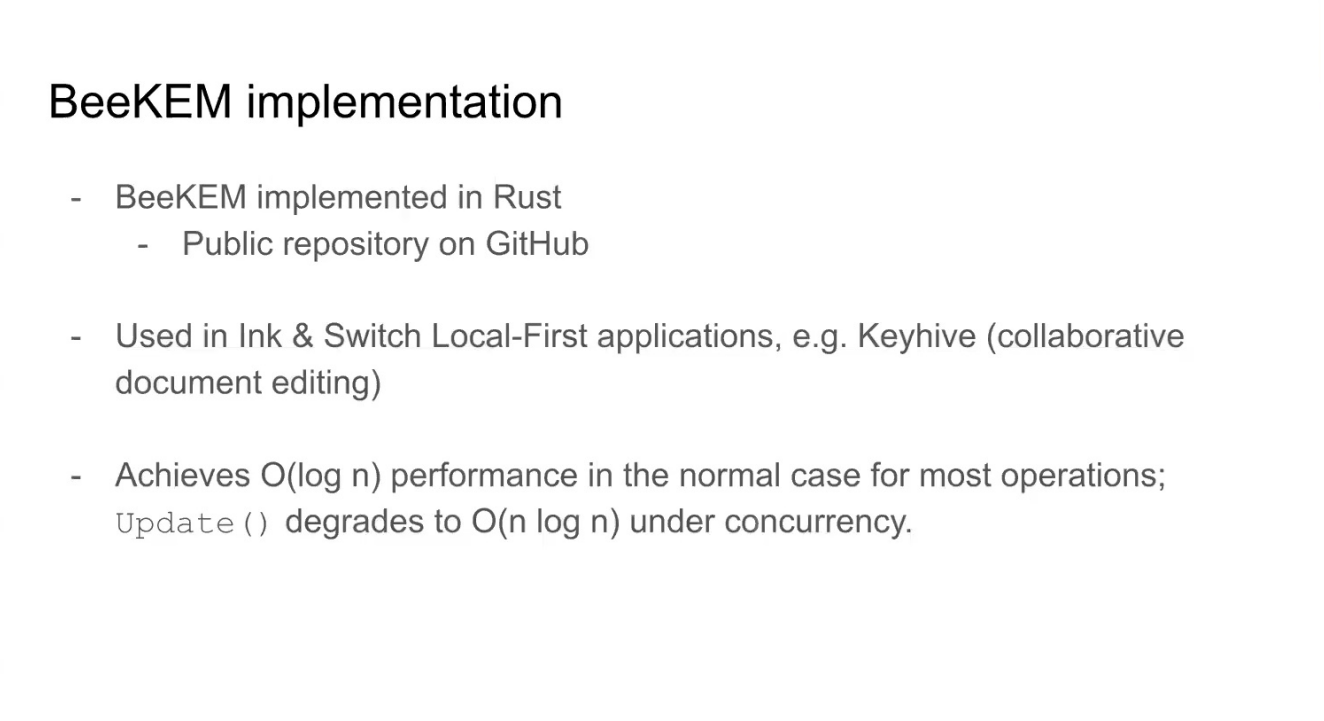
00:23:33 · Derek Yen

They cost only O(1). And when people update their public-private key pair, they also get to publish that as just one message. So what if we think about trying to improve encryption and instead do this thing where we will just distribute one symmetric secret and use that to encrypt, but we will encrypt that symmetric secret under each PK? Kind of a flat tree, basically. Encryption to the entire group will only cost O(1).

00:24:01 · Derek Yen

But the issue here is that changes to group membership, or to the secret, now take O(n), because you have to re-encrypt the new secret under every new public key. Or, sorry, under every existing public key. And if you add or remove a user, you need to rotate the group secret as well. So you can intuitively think of what a tree-based CGKA like BeeKEM, or prior to that, TreeKEM, as doing that middle solution in a slightly better way, where you achieve O(1) encryption but get logarithmic CGKA complexity by structuring it slightly differently.

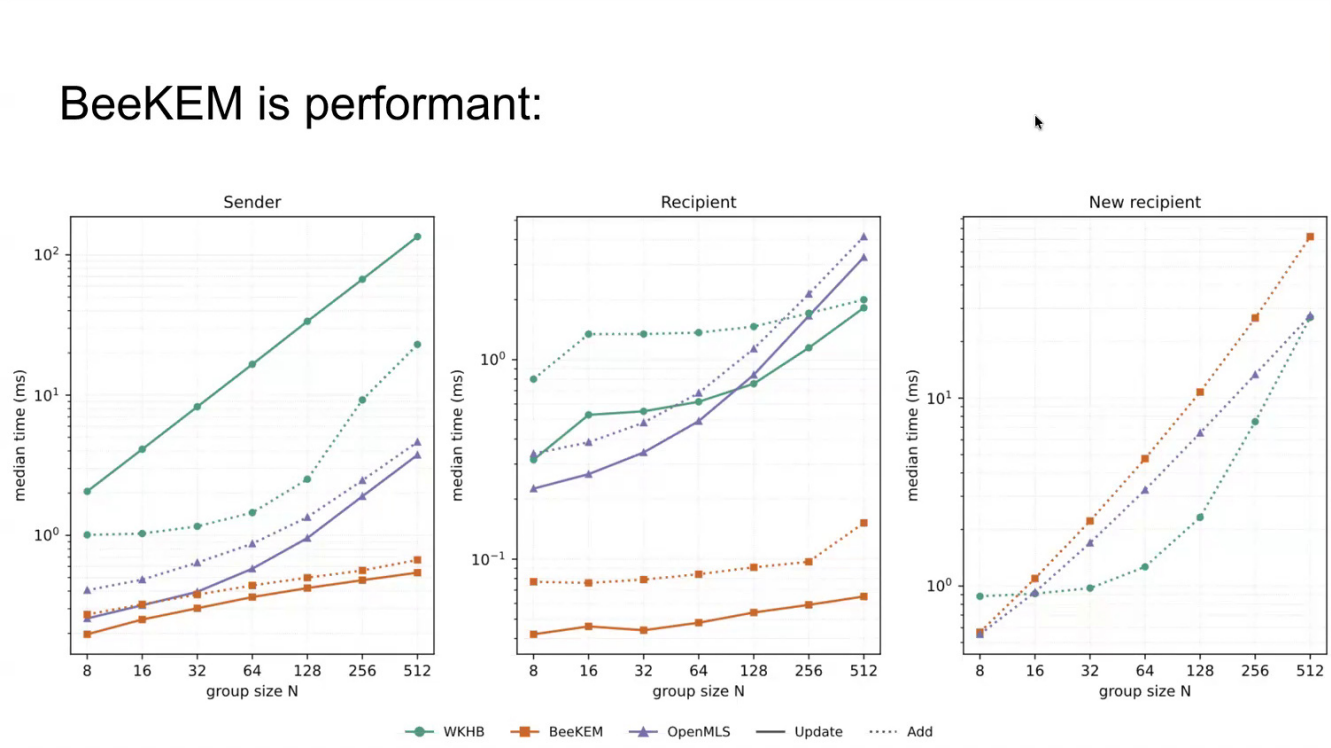
Rust implementation and performance



Implementation and concurrent updates · Frame 00:24:55

00:24:37 · Derek Yen

So now, lastly, I'll talk a little bit about our implementation of BeeKEM. So we have an implementation of BeeKEM in Rust, which has a public repository on GitHub, and it's already been used in some other applications, local-first applications, by Ink & Switch, such as their collaborative document editing application called Keyhive. BeeKEM has the asymptotic property of achieving logarithmic performance in the normal case. However, updates specifically can sometimes be more expensive due to concurrency, O(n log n).



BeeKEM, WKHB and OpenMLS performance · Frame 00:26:05

00:25:08 · Derek Yen

The details for that I can say offline, as it's a little bit more complicated. But that's the caveat I was alluding to earlier. I'll now show you some data of BeeKEM's performance. So here we have three plots showing on the y-axis the median time of performing operations, and on the x-axis the group size, so we can show kind of the asymptotic behavior of BeeKEM. We're plotting two different operations of CGKA schemes, update and add.

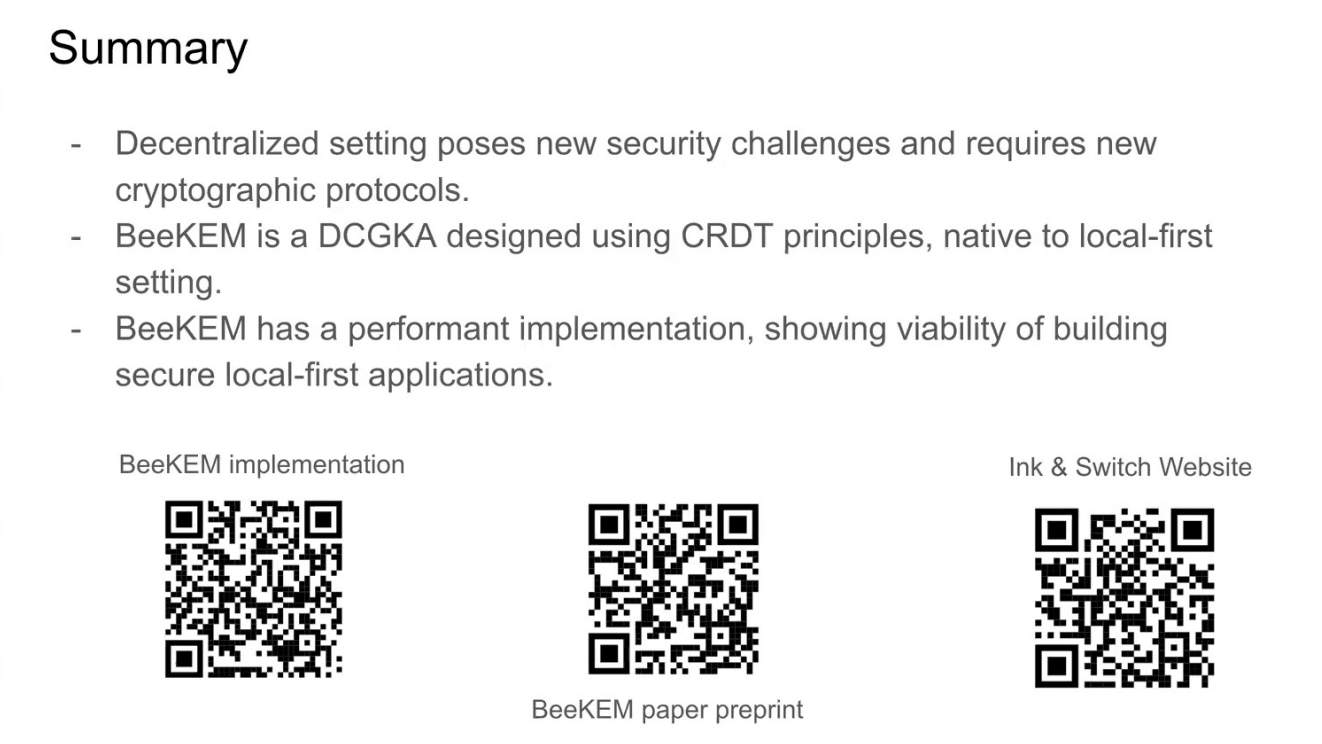
00:25:39 · Derek Yen

Update is the solid line, add is the dash line. And we're showing BeeKEM against two other representative CGKA schemes. So BeeKEM is the orange lines with the squares. We have WKHB, one of the prior state-of-the-art DCGKA schemes, in green. And we have OpenMLS, which is an, what's the word, an implementation of the MLS protocol, or TreeKEM, the CGKA, which is probably the most commonly used centralized CGKA scheme currently, in purple. So we see from these plots that BeeKEM is actually achieving performance on par with, or better, than these two other schemes in most regimes.

00:26:26 · Derek Yen

In the left and in the middle, we have the sender and the recipient costs respectively, which are different because sender and recipient have to perform different operations. And on the right, we have the cost associated with a new recipient joining the group, where we notice that BeeKEM is actually a bit worse, as you might expect, because the CRDT design philosophy requires you to store a lot more metadata. All right, so to summarize the talk, the decentralized setting poses new security challenges and requires new cryptographic protocols.

Summary and resources



Code, preprint and research group · Frame 00:27:30

00:26:56 · Derek Yen

Most cryptographic protocols have implicitly looked only at the centralized setting, and therefore cannot be used directly for local-first applications. BeeKEM, our DCGKA, has been designed using the principles of conflict-free replicated data types, and is therefore native to the local-first setting. BeeKEM has a performant implementation showing the viability of building secure local-first applications. And lastly, I have a couple of links. Here on the left, we have a QR code for the implementation of BeeKEM on GitHub.

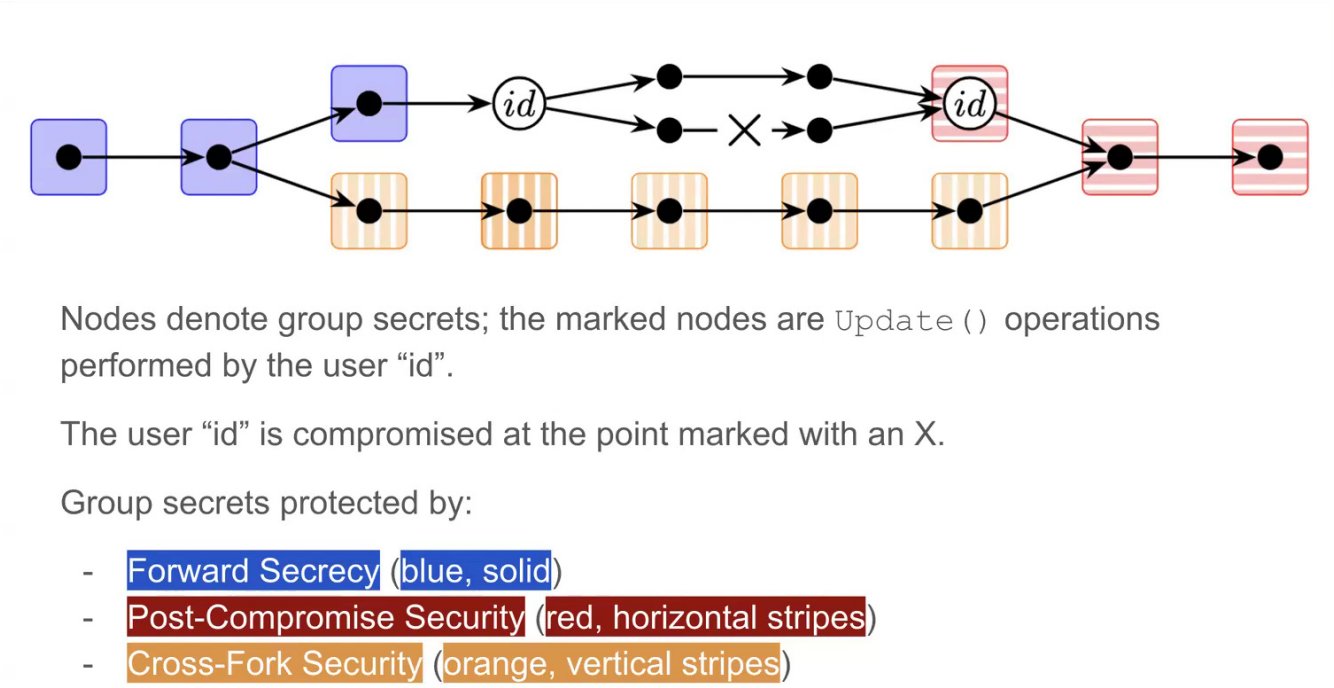
00:27:26 · Derek Yen

In the middle, we have the preprint for our paper explaining the algorithm of BeeKEM, proving its security and evaluating it, which is currently in submission. And on the right, we have a link to the Ink & Switch website, an industry group which does a lot of work for local-first, so if you haven't heard of them before, they're definitely worth checking out. So thank you so much for your time, and I'm free to take any questions.

Q&A: a cross-fork attack

00:27:48 · Host

You introduced the idea of cross-fork security. Could you give us a concrete example of what a cross-fork attack would look like? What could an attacker actually do if a protocol did not provide this property?



Security diagram used during Q&A · Frame 00:28:51

00:28:08 · Derek Yen

Yeah, that's a very good question. So I will give you a type of attack. So BeeKEM achieves a form of cross-fork security. I will give you an attack which works against the two prior DCGKA schemes, so the causal TreeKEM and the WKHB. So causal TreeKEM and WKHB work in pretty different ways, so I'll state the attack in a kind of generic way. If you have a situation where there's two concurrently defined updates, as in, like, let's say Alice and Bob are both members of a large group, and then Alice and Bob each concurrently publish updates, then there's two new group secrets, the one that Alice has defined and the one that Bob has defined.

00:28:54 · Derek Yen

For the two prior schemes, so the WKHB and for causal TreeKEM, if someone compromises Alice on this branch and obtains the group secret, they actually can obtain state that can be used to discover or compute Bob's new secret on this branch. That's stated kind of vaguely because the details matter a little bit for how it works in each of the schemes, but that would be an example of something that technically does not fall under either forward secrecy or post-compromise security, because intuitively these two new secrets are concurrent.

00:29:29 · Derek Yen

They don't have any relationship under the causal ordering. So technically speaking, neither forward secrecy or post-compromise security could apply. I think more broadly, what cross-fork security is trying to say is it, what's the right way to say this? I think maybe the most natural way to say it is that you do want there to be some kind of security claims you make about other branches under concurrency, and it's just something that isn't immediately implied by the two traditional security properties.

00:30:13 · Derek Yen

Let's see.

00:30:15 · Host

There's another question.

Q&A: security versus long offline periods

00:30:19 · Derek Yen

Yeah, okay. I'll read out the question for the recording. So what do you see as the biggest open problems in DCGKA? Where should the field go next? That's a good question. So one problem that I haven't really discussed here is, if you look at our preprint of BeeKEM, we technically achieve somewhat weaker security properties than the ones I state here. So the three security properties I've stated here are ideal security properties.

00:30:51 · Derek Yen

Like, the best possible DCGKA scheme would have these three properties. But what BeeKEM achieves is tuned actually with this parameter, which we denote kappa in the code, where essentially what this parameter does is it lets you show a kind of trade-off that exists between security and what we call a kind of, like, robustness to concurrency. So let me back out a little bit. Another property that you might want for a DCGKA scheme, or broadly, like a decentralized local-first scheme, is the ability to tolerate long partitions and long time of being offline.

00:31:30 · Derek Yen

So in particular, if I'm Alice and I'm in a group chat, and then I decide to go on vacation, going camping in the woods for three weeks, and I don't touch my phone. When I come back, I want to be able to still decrypt all of these messages that were meant for me. And it might be the case that if we're working in an application where we encrypt very large amounts of data, it would not be possible for them to then re-encrypt all of that data to send to me after I come back.

00:31:57 · Derek Yen

So I need a way to access the group secrets in those three weeks that I was gone. Achieving this property requires Alice to maintain a lot of prior versions of her secret, like very old secrets, individual secrets. And what this means is we weaken security a little bit, because if Alice's device were to be compromised, then the attacker would be able to obtain a lot of information, many prior group secrets. So kappa kind of denotes the trade-off between security and this kind of ability to tolerate partition and, like, long periods of downtime.

00:32:34 · Derek Yen

So finally, getting back to your question, one thing that we mention in this work is that we have this kind of trade-off in BeeKEM, at least, between this kind of usability under concurrency and security. But an ideal scheme would not have to make this trade-off. I think one thing we wanted to think about a little bit more is whether this trade-off is actually inherent to the setting, or if there are things you can do to combat it.

00:32:57 · Derek Yen

And we also define a variant of BeeKEM that makes the choice to achieve much stronger security, but at the cost of not tolerating concurrency as well. But maybe to wrap up my answer, I think we probably need a little bit more careful thinking about these two competing properties, especially for local-first. We really want these strong security properties, but local-first has certain practical engineering requirements as well for partition tolerance that are a little bit in tension with security, at least in our current understanding.

00:33:30 · Derek Yen

So I think that would be a very important question to look at with future research. Does that answer make sense? I think that was a bit of a long-winded answer.

Q&A: security state as a replicated data type

00:33:53 · Derek Yen

Yeah, reading out this question for the video: "Both BeeKEM and CRDTs have to deal with concurrent operations and merge them later. Do you think security state itself could eventually be modeled more like a replicated data type?" I think, to be honest, this question is a little bit beyond my expertise.

00:34:14 · Derek Yen

I'm coming more from the cryptography side and have been learning the CRDT and local-first things. I think I'm not sure.

Q&A: the origins of BeeKEM

00:34:25 · Host

How did you begin the research on BeeKEM? And what do you find most interesting and most fun about this work?

00:34:51 · Derek Yen

Yeah. So I've presented BeeKEM as this kind of standalone thing and mentioned that it's used in Ink & Switch's other application, Keyhive, but that's actually not the actual order in which things occurred. So at Ink & Switch, we developed first Keyhive, and to make this collaborative document editor, developed this DCGKA scheme. So technically, Keyhive came first. So that's kind of, like, the history of literally how BeeKEM came to be. The design philosophy really was coming from the perspective of CRDTs, as I understand it.

00:35:26 · Derek Yen

It really was the main idea here. Broadly, BeeKEM belongs to a family of CGKA protocols called ART protocols, asynchronous ratcheting tree, which is also the family that gives you TreeKEM, the most commonly used centralized CGKA. So that's kind of the intellectual history of BeeKEM. And I should also mention the first DCGKA scheme, causal TreeKEM, also was a very strong spiritual, a very strong, what's the word, ancestor of BeeKEM. In terms of, sorry, what was the second question again?

00:36:07 · Host

Which parts do you find most interesting for you, you find most exciting in this work?

Q&A: a cryptographer's surprising discovery

00:36:16 · Derek Yen

Yeah. I think for me personally, so as I mentioned before, I'm coming as a cryptographer into this space. Local-first is something new to me. For me, what was very interesting was realizing that there are a lot of ways that conventional cryptographic techniques are not relevant or do not apply directly to the local-first setting. So earlier when I said this thing about how end-to-end encryption is typically described as the server does nothing, so why does the server, or the server can't see anything, so why do we need the server?

00:36:49 · Derek Yen

When I first learned about BeeKEM and was developing the security proof with my collaborators at Ink & Switch, I was very surprised to realize that I had not recognized how end-to-end encryption actually relies on central servers. For me as a cryptographer, it was surprising to realize that this reliance still exists. It shows why the cryptographic community's interest in the role of the server really is important.

00:37:27 · Derek Yen

And I think that there's been that line of work trying to model the server and recognize the ways we assume certain properties of the server and challenge those. But I think that the local-first approach of completely removing the server is relatively understudied. So to me, that's a very interesting question.

Q&A: document history and forward secrecy

00:37:47 · Derek Yen

Reading out another question from the chat: "Documents seem different from chat. It seems that in the document setting we have to give up forward secrecy. Is that right?"

00:37:56 · Derek Yen

Yeah. So this is a conversation I've had with Brooke a couple of times. This is a very good question. I think it really depends on your application. And I'm not willing to say, like, in general, that documents should give up forward secrecy. But I think for most reasonable cases, yes, that's right. Forward secrecy does not make as much sense. To kind of explain this question a bit for the video, a very natural thing when you're doing collaborative document editing is that you want to see prior versions of the document.

00:38:33 · Derek Yen

At the very least, you need to see the current state of the document, which is a function of prior versions of the document. But you also usually want to see something more granular than that. And formally speaking, being able to see each of the individual edits made in the past to this document requires you, as a new member of the document, to be able to access the key that encrypted those prior edits.

00:38:56 · Derek Yen

In other words, you being added to the group and getting prior versions of the group secret requires you to lose a kind of forward secrecy, or to lose forward secrecy, I should say. I think that, to answer your question, if you want new users to see the full history of the group, then yes, you have to give up forward secrecy. But you could design it the other way, where you do have forward secrecy, but new users don't get to see the history of the group.

00:39:22 · Derek Yen

There might be applications where that's the trade-off that you prefer to make. But certainly, I think, if I can speak for Brooke, I think the perspective of Brooke and some other people who worked on Keyhive was that forward secrecy was not as important as users being able to see prior versions of the document history. And for Keyhive specifically, there are some other reasons why you give up forward secrecy, but that's specific to Keyhive.

00:39:51 · Derek Yen

Yeah, these are very good questions.

Q&A: building chat applications with BeeKEM

00:39:54 · Derek Yen

If someone built a chat app on BeeKEM, what interesting or fun features could it have?

00:40:00 · Derek Yen

Hmm. I think I might be the wrong person to ask this. I think the broad thing that's interesting about chat as a decentralized app is there are a lot of ways that messaging specifically is more okay with some of the constraints of the decentralized setting than other applications. So what I mean by that is a lot of, so taking the point that a key challenge of the decentralized setting is partition and concurrency, there are a lot of applications where if someone goes offline for a period of time, the application doesn't really work.

00:40:45 · Derek Yen

I'm trying to think of a suitable example. Suppose you have an application where every sent packet needs to be received, none can be dropped, and everyone needs to receive them, such as some kind of large joint computation. This type of application is difficult in the local-first setting because a partition might ruin the computation permanently and force you to start over.

00:41:24 · Derek Yen

This is the sort of thing where the centralized setting is a lot more natural. But what's nice about messages is that they have a kind of semantic robustness, as in, if I text someone and one of my messages is dropped, well, maybe that's a bad example, but at the very least you can detect that the message has been dropped, because maybe I say something that doesn't make sense. So chat specifically is a little bit more natural.

00:41:51 · Derek Yen

I think, yeah, I'm not sure if I have a general answer for the question, though. I think maybe kind of the goal here is that BeeKEM allows you to build any kind of chat app as if it were in the centralized setting, although the security and the properties are a little bit different. Sorry, that might not be a very good answer.