Detecting and preventing active attacks against Autocrypt

Release 0.10.0

NEXTLEAP researchers

Dec 22, 2018

Contents

1 Introduction

This document considers how to secure Autocrypt[1]-capable mail apps against active network attackers. Autocrypt aims to achieve convenient end-to-end encryption of e-mail. The Level 1 Autocrypt specification offers users opt-in e-mail encryption, but only considers passive adversaries. Active network adversaries, who could, for example, tamper with the Autocrypt header during e-mail message transport, are not considered in the Level 1 specification. Yet, such active attackers might undermine the security of Autocrypt. Therefore, we present and discuss new ways to prevent and detect active network attacks against Autocrypt[2]-capable mail apps.

We aim to help establish a *reverse panopticon*: a network adversary should not be able to determine whether peers discover malfeasant manipulations, or even whether they exchange information to investigate attacks. If designed and implemented successfully it means that those who (can) care for detecting malfeasance also help to secure the communications of others in the ecosystem.

This document reflects current research of the NEXTLEAP EU project. The NEXTLEAP project aims to secure Autocrypt beyond Level 1. To this end, this document proposes new Autocrypt protocols that focus on securely exchanging and verifying keys. To design these protocols, we considered usability, cryptographic and implementation aspects simultaneously, because they constrain and complement each other. Some of the proposed protocols are already implemented; we link to the repositories in the appropriate places.

Note: Future revisions are to refine this document and its contained protocols. The document lives at https://github.com/nextleap-project/countermitm and can be followed in public. A 1.0 release is envisioned for end 2018.

1.1 Attack model and terminology

We consider a *network adversary* that can read, modify, and create network messages. Examples of such an adversary are an ISP, an e-mail provider, an AS, or an eavesdropper on a wireless network. The goal of the adversary is to i) read the content of messages, ii) impersonate peers – communication partners, and iii) to learn who communicates with whom. To achieve these goals, an active adversary might try, for example, to perform a machine-in-the-middle attack on the key exchange protocol between peers. We consider this approach effective against mass surveillance of the encrypted email content while preventing additional meta data leakage.

[1] https://autocrypt.org
[2] https://autocrypt.org

To enable secure key-exchange and key-verification between peers, we assume that peers have access to a *out-of-band* communication channel that cannot be observed or manipulated by the adversary. More concretely we expect them to be able to transfer a small amount of data via a QR-code confidentially.

Targeted attacks on end devices or the out-of-band channels can break our assumptions and therefore the security properties of the protocols described. In particular the ability to observe QR-codes in the scan process (for example through CCTV or by getting access to print outs) will allow impersonation attacks. Additional measures can relax the security requirements for the *out-of-band* channel to also work under a threat of observation.

Passive attackers such as service providers can still learn who communicates with whom at what time and the approximate size of the messages. We recommend using additional meassures such as encrypting the subject to prevent further data leakage. This is beyond the scope of this document though.

Because peers learn the content of the messages, we assume that all peers are honest. They do not collaborate with the adversary and follow the protocols described in this document.

1.2 Problems of current key-verification techniques

An important aspect of secure end-to-end (e2e) encryption is the verification of a peer's key. In existing e2e-encrypting messengers, users perform key verification by triggering two fingerprint verification workflows: each of the two peers shows and reads the other's key fingerprint through a trusted channel (often a QR code show+scan).

We observe the following issues with these schemes:

- The schemes require that both peers start the verification workflow to assert that both of their encryption keys are not manipulated. Such double work has an impact on usability.
- In the case of a group, every peer needs to verify keys with each group member to be able to assert that messages are coming from and are encrypted to the true keys of members. A peer that joins a group of size N must perform N verifications. Forming a group of size N therefore requires $N(N-1)/2$ verifications in total. Thus this approach is impractical even for moderately sized groups.
- The verification of the fingerprint only checks the current keys. Since protocols do not store any historical information about keys, the verification can not detect if there was a past temporary MITM-exchange of keys (say the network adversary exchanged keys for a few weeks but changed back to the "correct" keys afterwards).
- Users often fail to distinguish Lost/Reinstalled Device events from Machine-in-the-Middle (MITM) attacks, see for example When Signal hits the Fan[3].

[3] https://eurousec.secuso.org/2016/presentations/WhenSignalHitsFan.pdf

1.3 Integrating key verification with general workflows

In *Securing communications against network adversaries* (page 6) we describe new protocols that aim to resolve these issues, by integrating key verification into existing messaging use cases:

- the *Setup Contact protocol* (page 7) allows a user, say Alice, to establish a verified contact with another user, say Bob. At the end of this protocol, Alice and Bob know each other's contact information and have verified each other's keys. To do so, Alice sends bootstrap data using the trusted out-of-band channel to Bob (for example, by showing QR code). The bootstrap data transfers not only the key fingerprint, but also contact information (e.g., email address). After receiving the out-of-band bootstrap data, Alice's and Bob's clients communicate via the regular channel to 1) exchange Bob's key and contact information and 2) to verify each other's keys. Note that this protocol only uses one out-of-band message requiring involvement of the user. All other messages are transparent.

- the *Verified Group protocol* (page 12) enables a user to invite another user to join a verified group. The "joining" peer establishes verified contact with the inviter, and the inviter then announces the joiner as a new member. At the end of this protocol, the "joining" peer has learned the keys of all members of the group. This protocol builds on top of the previous protocol. But, this time, the bootstrap data functions as an invite code to the group.

  Any member may invite new members. By introducing members in this incremental way, a group of size N requires only $N - 1$ verifications overall to ensure that a network adversary can not compromise end-to-end encryption between group members. If one group member loses her key (e.g. through device loss), she must re-join the group via invitation of the remaining members of the verified group.

- the *History verification protocol* (page 17) verifies the cryptograhic integrity of past messages and keys. It can precisely point to messages where cryptographic key information has been modified by the network.

Moreover, in *Securing communications against network adversaries* (page 6) we also discuss a privacy issue with the Autocrypt Key gossiping mechanism. The continuous gossipping of keys may enable an observer to infer who recently communicated with each other. We present an "onion-key-lookup" protocol which allows peers to verify keys without other peers learning who is querying a key from whom. Users may make onion key lookups to learn and verify key updates from group members: if a peer notices inconsistent key information for a peer it can send an onion-key query to resolve the inconsistency.

Onion key lookups also act as cover traffic which make it harder for the network to know which user is actually communicating with whom.

1.4 Supplementary key consistency through ClaimChains

We discuss a variant of ClaimChain[4], a distributed key consistency scheme, in which all cryptographic checks are performed on the end-point side. ClaimChains are self-authenticated hash chains whose blocks contain statements about key material of the ClaimChain owner and the key material of her contacts. The "head" of the ClaimChain, the latest block, represents a commitment to the current state, and the full history of past states.

ClaimChain data structures track all claims about public keys and enable other peers to automatically verify the integrity of claims. ClaimChains include cryptographic mechanisms to ensure the *privacy of the claim it stores* and the *privacy of the user's social graph*. Only authorized users can access the key material and the cross-references being distributed. In other words, neither providers nor unauthorized users can learn anything about the key material in the ClaimChain and the social graph of users by just observing the data structure.

Private claims could be used by malicious users (or a network adversary who impersonates users) to *equivocate*, i.e., present a different view of they keys they have seen to their peers. For example, Alice could try to equivocate by showing different versions of a cross-reference of Bob's key to Carol and Donald. Such equivocations would hinder the ability to resolve correct public keys. Therefore, ClaimChain prevents users (or a network adversaries) from *equivocating* to other users about their cross-references.

The implementation of ClaimChains considered in this document relies on a self-authenticating storage which, given a hash, replies with a matching data block. We suggest that providers provide a "dumb" block storage for their e-mail customers, re-using existing authentication techniques for guarding writes to the block storage. The head hashes that allow to verify a full chain are distributed along with Autocrypt Gossip headers. Given a head, peers can verify that a chain has not been tampered with and represents the latest belief of another peer. Peers can use the information in the chain to perform consistency checks.

ClaimChain permits users to check the evolution of others' keys over time. If inspection of the Claimchains reveals inconsistencies in the keys of a peer – for example, because an adversary tampered with the keys – the AutoCrypt client can advice the user to run the *History-verification protocol* (page 17) with this inconsistent peer. This protocol will then reveal conclusive evidence of malfeasance.

1.5 Detecting inconsistencies through Gossip and DKIM

The protocols for key verification and key inconsistency aid to detect malfeasance. However, even if they were not added, mail apps can use existing Autocrypt Level 1 Key Gossip and DKIM signatures to detect key inconsistencies.

Key inconsistencies or broken signatures found using these methods can not be interpreted unequivocally as proof of malfeasance. Yet, mail apps can track such events and provide recommen-

[4] https://claimchain.github.io/

dations to users about “Who is the most interesting peer to verify keys with?” so as to detect real attacks.

We note that if the adversary isolates a user by consistently injecting MITM-keys on her communications, the adversary can avoid the “inconsistency detection” via Autocrypt’s basic mechanisms. However, any out-of-band key-history verification of that user will result in conclusive evidence of malfeasance.

2 Securing communications against network adversaries

To withstand network adversaries, peers must verify each other's keys to establish trustable e2e-encrypted communication. In this section we describe protocols to securely setup a contact, to securely add a user to a group, and to verify key history.

Establishing a trustable e2e-encrypted communication channel is particularly difficult in group communications where more than two peers communicate with each other. Existing messaging systems usually require peers to verify keys with every other peer to assert that they have a trustable e2e-encrypted channel. This is highly unpractical. First, the number of verifications that a single peer must perform becomes too costly even for small groups. Second, a device loss will invalidate all prior verifications of a user. Rejoining the group with a new device (and a new key) requires redoing all the verification, a tedious and costly task. Finally, because key verification is not automatic – it requires users' involvement – in practice very few users consistently perform key verification.

Key consistency schemes do not remove the need of key verification. It is possible to have a group of peers which each see consistent email-addr/key bindings from each other, yet a peer is consistently isolated by a network adversary performing a machine-in-the-middle attack. It follows that each peer needs to verify with at least one other peer to assure that there is no isolation attack.

A known approach to reduce the number of neccessary key verifications is the web of trust. This approach requires a substantial learning effort for users to understand the underlying concepts, and is hardly used outside specialist circles. Moreover, when using OpenPGP, the web of trust is usually interacting with OpenPGP key servers. These servers make the signed keys widely available, effectively making the social "trust" graph public. Both key servers and the web of trust have reached very limited adoption.

Autocrypt was designed to not rely on public key servers, nor on the web of trust. It thus provides a good basis to consider new key verification approaches. To avoid the difficulties around talking about keys with users, we suggest new protocols which perform key verification as part of other workflows, namely:

- setting up a contact between two individuals who meet physically, and
- setting up a group with people who you meet or have met physically.

These new workflows require *administrative* messages to support the authentication and security of the key exchange process. These administrative messages are sent between devices, but are not shown to the user as regular messages. This is a challenge, because some e-mail apps display all messages (including machine-generated ones for rejected or non-delivered mails) without special rendering of the content. Only some messengers, such as Delta-chat[5], already use administrative messages, e.g., for group member management.

The additional advantage of using administrative messages is that they significantly improve usability by reducing the overall number of actions to by users. In the spirit of the strong UX focus of

[5] https://delta.chat

the Autocrypt specification, however, we suggest to only exchange administrative messages with peers when there there is confidence they will not be displayed "raw" to users, and at best only send them on explicit request of users.

Note that automated processing of administrative messages opens up a new attack vector: malfeasant peers can try to inject adminstrative messages in order to impersonate another user or to learn if a particular user is online.

All protocols that we introduce in this section are *decentralized*. They describe how peers (or their devices) can interact with each other, without having to rely on services from third parties. Our verification approach thus fits into the Autocrypt key distribution model which does not require extra services from third parties either.

Autocrypt Level 1 focusses on passive attacks such as sniffing the mail content by a provider. Active attacks are outside of the scope and can be carried out automatically by replacing Autocrypt headers.

Here we aim to increase the costs of active attacks by introducing a second channel and using it to verify the Autocrypt headers transmitted in-band.

We consider targeted active attacks against these protections feasible. However they will require coordinated attacks based for example on infiltrators or real time CCTV footage.

We believe that the ideas explained here make automated mass surveillance prohibitively expensive with a fairly low impact on usability.

2.1 Setup Contact protocol

The goal of the Setup Contact protocol is to allow two peers to conveniently establish secure contact: exchange both their e-mail addresses and cryptographic identities in a verified manner. This protocol is re-used as a building block for the *history-verification* (page 17) and *verified-group* (page 12) protocols.

After running the Setup Contact protocol, both peers will learn the cryptographic identities (i.e., the keys) of each other or else both get an error message. The protocol is safe against active attackers that can modify, create and delete messages.

The protocol follows a single simple UI workflow: A peer "shows" bootstrap data that is then "read" by the other peer through a second channel. This means that, as opposed to current fingerprint verification workflows, the protocol only runs once instead of twice, yet results in the two peers having verified keys of each other.

Between mobile phones, showing and scanning a QR code constitutes a second channel, but transferring data via USB, Bluetooth, WLAN channels or phone calls is possible as well.

Recall that we assume that our active attacker *cannot* observe or modify data transferred via the second channel.

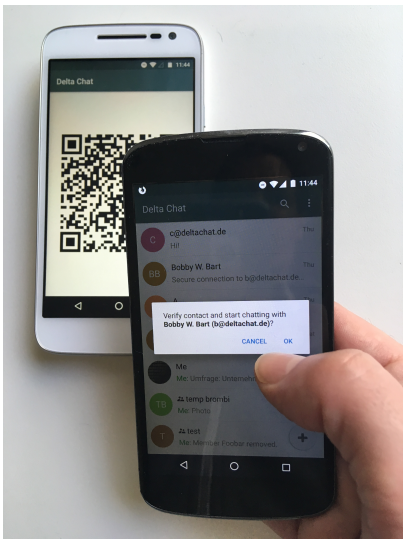

Fig. 1: Setup Contact protocol step 2 with https://delta.chat.

An attacker who can alter messages but has no way of reading or manipulating the second channel can prevent the verification protocol from completing successfully by droping or altering messages.

An attacker who can compromise both channels can inject wrong key material and convince the peer to verify it.

Fig. 2: UI and administrative message flow of contact setup

Here is a conceptual step-by-step example of the proposed UI and administrative message workflow for establishing a secure contact between two contacts, Alice and Bob.

1. Alice sends a bootstrap code to Bob via the second channel.

   (a) The bootstrap code consists of:

   - Alice's Openpgp4 public key fingerprint `Alice_FP`, which acts as a commitment to the Alice's Autocrypt key, which she will send later in the protocol,
   - Alice's e-mail address (both name and routable address),
   - A type `TYPE=vc-invite` of the bootstrap code
   - a challenge `INVITENUMBER` of at least 8 bytes. This challenge is used by Bob's device in step 2b to prove to Alice's device that it is the device that the bootstrap code was shared with. Alice's device uses this information in step 3 to automatically accept Bob's contact request. This is in contrast with most messaging apps where new contacts typically need to be manually confirmed.
   - a second challenge `AUTH` of at least 8 bytes which Bob's device uses in step 4 to authenticate itself against Alice's device.
   - optionally add metadata such as `INVITE-TO=groupname`

b) Per INVITENUMBER Alices device will keep track of: - the associated AUTH secret - the time the contact verification was initiated. - the metadata provided.

2. Bob receives the bootstrap code and

   (a) If Bob's device already knows a key with the fingerprint Alice_FP that belongs to Alice's e-mail address the protocol continues with 4b)

   (b) otherwise Bob's device sends a cleartext "vc-request" message to Alice's e-mail address, adding the INVITENUMBER from step 1 to the message. Bob's device automatically includes Bob's AutoCrypt key in the message.

3. Alice's device receives the "vc-request" message.

   a) She looks up the bootstrap data for the INVITENUMBER. If the INVITENUMBER does not match then Alice terminates the protocol.

   b) If she recognizes the INVITENUMBER from step 1 she checks that the invite has not expired. If the timestamp associated with the INVITENUMBER is longer ago than a given time Alice terminates the protocol.

   (c) She then processes Bob's Autocrypt key.

   d) She uses this key to create an encrypted "vc-auth-required" message containing her own Autocrypt key, which she sends to Bob.

4. Bob receive the "vc-auth-required" message, decrypts it, and verifies that Alice's Autocrypt key matches Alice_FP.

   (a) If verification fails, Bob gets a screen message "Error: Could not setup a secure connection to Alice" and the protocol terminates.

   (b) Otherwise Bob's device sends back a 'vc-request-with-auth' encrypted message whose encrypted part contains Bob's own key fingerprint Bob_FP and the second challenge AUTH from step 1.

5. Alice decrypts Bob's 'vc-request-with-auth' message

   a) and verifies that Bob's Autocrypt key matches Bob_FP that the invite has not expired and that the transferred AUTH matches the one from step 1.

   b) If any verification fails, Alice's device signals "Could not establish secure connection to Bob" and the protocol terminates.

6. If the verification succeeds on Alice's device

   (a) shows "Secure contact with Bob <bob-adr> established".

   (b) sends Bob a "vc-contact-confirm" message.

   (c) also removes the data associated with INVITECODE.

7. Bob's device receives "vc-contact-confirm" and shows "Secure contact with Alice <alice-adr> established".

At the end of this protocol, Alice has learned and validated the contact information and Autocrypt key of Bob, the person to whom she sent the bootstrap code. Moreover, Bob has learned and validated the contact information and Autocrypt key of Alice, the person who sent the bootstrap code to Bob.

2.1.1 Requirements for the underlying encryption scheme

The Setup Contact protocol requires that the underlying encryption scheme is non-malleable. Malleability means the encrypted content can be changed in a deterministic way. Therefore with a malleable scheme an attacker could impersonate Bob: They would add a different autocrypt key in Bob's vc-request message (step 2.b) and send the message along without other changes. In step 4.b they could then modify the encrypted content to include their own keys fingerprint rather than `Bob_FP`.

In the case of OpenPGP non-malleability is achieved with Modification Detection Codes (MDC - see section 5.13 and 5.14 of RFC 4880). Implementers need to make sure to verify these and treat invalid or missing MDCs as an error. Using an authenticated encryption scheme prevents these issues and is therefore recommended if possible.

2.1.2 An active attacker cannot break the security of the Setup Contact protocol

Recall that an active attacker can read, modify, and create messages that are sent via a regular channel. The attacker cannot observe or modify the bootstrap code that Alice sends via the second channel. We argue that such an attacker cannot break the security of the Setup Contact protocol, that is, the attacker cannot impersonate Alice to Bob, or Bob to Alice.

Assume, for a worst-case scenario, that the adversary knows the public Autocrypt keys of Alice and Bob. At all steps except step 1, the adversary can drop messages. Whenever the adversary drops a message, the protocol fails to complete. Therefore, we do not consider dropping of messages further.

1. The adversary cannot impersonate Alice to Bob, that is, it cannot replace Alice's key with a key Alice-MITM known to the adversary. Alice sends her key to Bob in the encrypted "vc-auth-required" message (step 3). The attacker can replace this message with a new "vc-auth-required" message, again encrypted against Bob's real key, containing a fake Alice-MITM key. However, Bob will detect this modification step 4a, because the fake Alice-MITM key does not match the fingerprint `Alice_FP` that Alice sent to Bob in the bootstrap code. (Recall that the bootstrap code is transmitted via the second channel the adversary cannot modify.)
2. The adversary also cannot impersonate Bob to Alice, that is, it cannot replace Bob's key with a key Bob-MITM known to the adversary. The cleartext "vc-request" message, sent from Bob to Alice in step 2, contains Bob's key. To impersonate Bob, the adversary must substitute this key with the fake Bob-MITM key.

In step 3, Alice cannot distinguish the fake key Bob-MITM inserted by the adversary from Bob's real key, since she has not seen Bob's key in the past. Thus, she will follow the protocol and send the reply "vc-auth-required" encrypted with the key provided by the adversary.

We saw in the previous part that if the adversary modifies Alice's key in the "vc-auth-required" message, then this is detected by Bob. Therefore, it forwards the "vc-auth-required" message unmodified to Bob.

Since `Alice_FP` matches the key in "vc-auth-required", Bob will in step 4b send the "vc-request-with-auth" message encrypted to Alice's true key. This message contains Bob's fingerprint `Bob_FP` and the challenge `AUTH`.

Since the message is encrypted to Alice's true key, the adversary cannot decrypt the message to read its content. There are now three possibilities for the attacker:

- The adversary modifies the "vc-request-with-auth" message to replace `Bob_FP` (which it knows) with the fingerprint of the fake Bob-MITM key. However, the encryption scheme is non-malleable, therefore, the adversary cannot modify the message, without being detected by Alice.
- The adversary drops Bob's message and create a new fake message containing the finger print of the fake key Bob-MITM and a guess for the challenge `AUTH`. The adversary cannot learn the challenge `AUTH`: it cannot observe the bootstrap code transmitted via the second channel in step 1, and it cannot decrypt the message "vc-request-with-auth". Therefore, this guess will only be correct with probability 2^{-64}. Thus, with overwhelming probability Alice will detect the forgery in step 5, and the protocol terminates without success.
- The adversary forwards Bob's original message to Alice. Since this message contains Bob's key fingerprint `Bob_FP`, Alice will detect in step 5 that Bob's "vc-request" from step 3 had the wrong key (Bob-MITM) and the protocol terminates with failure.

2.1.3 Replay attacks and conflicts

Alices device records the time a contact verification was initiated. It also verifies it has not expired and clears the data after completion. This prevents replay attacks. Replay attacks could be used to make Alices device switch back to an old compromised key of Bob.

Limiting an invite to a single use reduces the impact of a QR-code being exposed to an attacker: If the attacker manages to authenticate faster than Bob they can impersonate Bob to Alice. However Bob will see an error message. If the QR-code could be reused the attacker could successfully authenticate. Alice would have two verified contacts and Bob would not see any difference to a successful connection attempt.

Furthermore a compromise of Bob's device would allow registering other email addresses as verified contacts with Alice.

2.1.4 Business Cards

QR-codes similar to the ones used for verified contact could be used to print on business cards.

Since business cards are usually not treated as confidential they can only serve to authenticate the issuer of the business card (Alice) and not the recipient (Bob).

However as discussed on the messaging@moderncrypto mailing list[6] the verification of a short code at the end of the protocol can extend it to also protect against leakage of the QR-code. This may also be desirable for users who face active surveillance in real life and therefor cannot assume that scanning the QR-code is confidential.

2.1.5 Open Questions

- (how) can messengers such as Delta.chat make "verified" and "opportunistic" contact requests be indistinguishable from the network layer?
- (how) could other mail apps such as K-9 Mail / OpenKeychain learn to speak the "setup contact" protocol?

2.2 Verified Group protocol

We introduce a new secure **verified group** that enables secure communication among the members of the group. Verified groups provide these simple to understand properties:

1. All messages in a verified group are end-to-end encrypted and secure against active attackers. In particular, neither a passive eavesdropper, nor an attactive network attacker (e.g., capable of man-in-the-middle attacks) can read or modify messages.
2. There are never any warnings about changed keys (like in Signal) that could be clicked away or cause worry. Rather, if a group member loses her device or her key, then she also looses the ability to read from or write to the verified group. To regain access, this user must join the group again by finding one group member and perform a "secure-join" as described below.

2.2.1 Verifying a contact to prepare joining a group

The goal of the secure-join protocol is to let Alice make Bob a member (i.e., let Bob join) a verified group of which Alice is a member. Alice may have created the group or become a member prior to the addition of Bob.

In order to add Bob to the group Alice has to verify him as a contact if she has not done so yet. We use this message exchange to also ask Bob wether he agrees to becoming part of the group.

[6] https://moderncrypto.org/mail-archive/messaging/2018/002544.html

The protocol re-uses the first five steps of the *setup-contact* (page 7) protocol so that Alice and Bob verify each other's keys. To ask for Bob's explicit consent we indicate that the messages are part of the verified group protocol, and include the group's identifier in the metadata part of the bootstrap code.

More precisely:

- in step 1 Alice adds the metadata `INVITE=<groupname>`. Where `<groupname>` is the name of the group `GROUP`.
- in step 2 Bob manually confirms he wants to join `GROUP` before his device sends the `vc-request` message. If Bob declines processing aborts.
- in step 5 Alice looks up the metadata associated with the `INVITENUMBER`. If Alice sees the `INVITE=<groupname>` but is not part of the group anymore she aborts the joining process (without sending another message).

If no failure occurred up to this point, Alice and Bob have verified each other's keys, and Alice knows that Bob wants to join the group `GROUP`.

The protocol then continues as described in the following section (steps 6 and 7 of the *setup-contact* (page 7) are not used).

2.2.2 Joining a verified group ("secure-join")

In order to add Bob to a group Alice first needs to make sure she has a verified key for Bob. This is the case if Bob already was a verified contact or Alice performed the steps described in the previous section.

Now she needs to inform the group that Bob should be added. Bob needs to confirm everything worked:

1. Alice broadcasts an encrypted "vg-member-setup" message to all members of `GROUP` (including Bob), gossiping the Autocrypt keys of all members (including Bob).
2. Bob receives the encrypted "vg-member-setup" message. Bob's device verifies:
   - The encryption and Alices signature are intact.
   - Alice may invite Bob to a verified group. That is she is a verified contact of Bob.

   If any of the checks fail processing aborts. Otherwise the device learns all the keys and e-mail addresses of group members. Bob's device sends a final "vg-member-setup-received" message to Alice's device. Bob's device shows "You successfully joined the verified group `GROUP`".
3. Any other group member that receives the encrypted "vg-member-setup" message will process the gossiped key through autocrypt gossip mechanisms. In addition they verify:
   - The encryption and Alices signature are intact.

- They are themselves a member of `GROUP`.
- Alice is a member of `GROUP`.

If any of the checks fail processing aborts. Otherwise they will add Bob to their list of group members and mark the gossiped key as verified in the context of this group.

4. Alice's device receives the "vg-member-setup-received" reply from Bob and shows a screen "Bob <email-address> securely joined group `GROUP`"

Bob and Alice may now both invite and add more members which in turn can add more members. The described secure-join workflow guarantees that all members of the group have been verified with at least one member. The broadcasting of keys further ensures that all members are fully connected.

Fig. 3: Join-Group protocol at step 2 with https://delta.chat.

2.2.3 Strategies for verification reuse

Since we retrieve keys for verified groups from peers we have to choose wether we want to trust our peers to verify the keys correctly.

One of the shortcomings of the web of trust is that it's mental model is hard to understand and make practical use of. We therefore do not ask the user questions about how much they trust their peers.

Therefore two strategies remain that have different security implications:

- **Restricting verification reuse accross groups** Since we share the content of the group with all group members we can also trust them to verify the keys used for the group.

If they wanted to leak the content they could do so anyway.

However if we want to reuse keys from one verified group to form a different one the peer who originally verified the key may not be part of the new group.

If the verifier is "malicious" and colludes with an attacker in a MITM position, they can inject a MITM key as the verified key. Reusing the key in the context of another group would allow MITM attacks on that group.

This can be prevented by restricting the invitation to verified groups to verified contacts and limiting the scope of keys from member-added messages to the corresponding group.

- **Ignoring infiltrators, focusing on message transport attacks first** One may also choose to not consider advanced attacks in which an "infiltrator" peer collaborates with an evil provider to intercept/read messages.

  In this case keys can be reused accross verified groups. Active attacks from an adversary who can only modify messages in the first channel are still impossible.

  A malicious verified contact may inject MITM keys. Say Bob when adding Carol as a new member, sends a prepared MITM key. We refer to this as a Bob in the middle attack to illustrate that a peer is involved in the attack.

  We note, that Bob, will have to sign the message containing the gossip fake keys. In the following section we introduce *history verification* which will detect such attacks after the fact. Performing a history verification with Alice will inform Carol about the MITM key introduced by Bob. Bob's signature serves as evidence that Bob gossiped the wrong key for Alice.

  Trusting all peers to verify keys also allows faster recovery from device loss. Say Alice lost her device and Bob verified the new key. Once Bob announced the new key in a verified group including Carol Carol could send the key to further verified groups that Bob is not part of.

2.2.4 Dealing with key loss and compromise

If a user looses their device they can setup a new device and regain access to their inbox. However they may loose their secret key.

They can generate a new key pair. Autocrypt will distribute their new public key in the Autocrypt headers and opportunistic encryption will switch to it automatically.

Verified groups will remain unreadable until the user verifies a contact from that group. Then the contact can update the key used in the group. This happens by sending a "vg-member-setup" message to the group. Since the email address of that user remains the same the old key will be replaced by the new one.

Implementers may decide wether the recipients of such key updates propagate them to other groups they share with the user in question. If they do this will speed up the recovery from device loss.

However it also allows Bob-in-the-middle attacks that replace the originally verified keys. So the decision needs to be based on the threat model of the app and the strategy picked for verification reuse

If a key is known or suspected to be compromised more care needs to be taken. Since network attackers can drop messages they can also drop the "vg-member-setup" message that was meant to replace a compromised key. A compromised key combined with a network attack breaks the security of both channels. Recovering from this situation needs careful consideration and goes beyond the scope of our current work.

2.2.5 Notes on the verified group protocol

- **More Asynchronous UI flow**: All steps after 2 (the sending of adminstrative messages) could happen asynchronously and in the background. This might be useful because e-mail providers often delay initial messages ("greylisting") as mitigation against spam. The eventual outcomes ("Could not establish verified connection" or "successful join") can be delivered in asynchronous notifications towards Alice and Bob. These can include a notification "verified join failed to complete" if messages do not arrive within a fixed time frame. In practise this means that secure joins can be concurrent. A member can show the "Secure Group invite" to a number of people. Each of these peers scans the message and launches the secure-join. As 'vc-request-with-auth' messages arrive to Alice, she will send the broadcast message that introduces every new peer to the rest of the group. After some time everybody will become a member of the group.
- **Leaving attackers in the dark about verified groups**. It might be feasible to design the step 3 "secure-join-requested" message from Bob (the joiner) to Alice (the inviter) to be indistinguishable from other initial "contact request" messages that Bob sends to Alice to establish contact. This means that the provider would, when trying to substitute an Autocrypt key on a first message between two peers, run the risk of **immediate and conclusive detection of malfeasance**. The introduction of the verified group protocol would thus contribute to securing the e-mail encryption eco-system, rather than just securing the group at hand.
- **Sending all messages through alternative channels**: instead of being relayed through the provider, all messages from step 2 onwards could be transferred via Bluetooth or WLAN. This way, the full invite/join protocol would be completed on a different channel. Besides increasing the security of the joining, an additional advantage is that the provider would not gain knowledge about verifications.
- **Non-messenger e-mail apps**: instead of groups, traditional e-mail apps could possibly offer the techniques described here for "secure threads".

2.2.6 Autocrypt and verified key state

Verified key material – whether from verified contacts or verified groups – provides stronger security guarantees then keys discovered in Autocrypt headers.

At the same time opportunistic usage of keys from autocrypt headers provides faster recovery from device loss.

Therefore the address-to-key mappings obtained using the verification protocols should be stored separately and in addition to the data stored for the normal Autocrypt behaviour.

Verified contacts and groups offer a separate communication channel from the opportunistic one.

We separated the two concepts but they can both be presented to the user as 'Verified Groups'. In this case the verified contact is a verified group with two members.

This allows the UI to feature a verified group and the 'normal' opportunistic encryption with the same contact.

The verified group prevents key injection through Autocrypt headers. In the case of device loss the user can fall back to the non-verified contact to ensure availability of a communication channel even before the next verification has taken place.

2.3 History-verification protocol

The two protocols we have described so far assure the user about the validity of the keys they verify and of the keys of their peers in groups they join. If the protocols detect an active attack (for example because keys are substituted) they immediately alert the user. Since users are involved in a verification process, this is the right time to alert users. By contrast, today's verification workflows alert the users when a previously key has changed. At that point users typically are not physically next to each other, and are rarely concerned with the key since they want to get a different job done, e.g., of sending or reading a message.

However, our new verification protocols only verify the current keys. Historical interactions between peers may involve keys that have never been verified using these new verification protocols. So how can users determine the integrity of keys of historical messages? This is where the history-verification protocol comes in. This protocol, that again relies on a second channel, enables two peers to verify integrity, authenticity and confidentiality of their shared historic messages. After completion, users gain assurance that not only their current communication is safe but that their past communications have not been compromised.

By verifying all keys in the shared history between peers, the history-verification protocol can detect temporary malfeasant substitutions of keys in messages. Such substitutions are not caught by current key-fingerprint verification workflows, because they only provide assurance about the current keys. They can detect substitutions that happened via gossip, Autocrypt headers and through verification reuse (Bob in the middle attacks).

In the latter case they also point out and provide evidence who introduced the MITM key in a given group. Performing a history verification with that person will in turn show where they got the key from. This way the key can be tracked back to who originally created it.

Like in the *setup-contact* (page 7) protocol, we designed our history-verification protocol so that peers only perform only one "show" and "read" of bootstrap information (typically transmitted via

showing QR codes and scanning them).

The protocol re-uses the first five steps of the *setup-contact* (page 7) protocol so that Alice and Bob verify each other's keys. We make one small modifications to indicate that the messages are part of the history-verification protocol: In step 1 Alice adds the metadata `VERIFY=history`.

If no failure occurred after step 5, Alice and Bob have again verified each other's keys. The protocol then continues as follows (steps 6 and 7 of the *setup-contact* (page 7) are not used):

6. Alice and Bob have each others verified Autocrypt key. They use these keys to encrypt a message to the other party which contains a **message/keydata list**. For each message that they have exchanged in the past they add the following information:
   - The message id of that message
   - When this message was sent, i.e., the `Date` field.
   - A list of (email-address, key fingerprints) tuples which they sent or received in that particular message.
7. Alice and Bob independently perform the following history-verification algorithm:
   (a) determine the start-date as the date of the earliest message (by `Date`) for which both sides have records.
   (b) verify the key fingerprints for each message since the start-date for which both sides have records of: if a key differs for any e-mail address, we consider this is strong evidence that there was an active attack. If such evidence is found, an error is shown to both Alice and Bob: "Message at <DATE> from <From> to <recipients> has mangled encryption".
8. Alice and Bob are presented with a summary which lists:
   - time frame of verification
   - the number of messages successfully verified
   - the number of messages with mangled encryption
   - the number of dropped messages, i.e. sent by one party, but not received by the other, or vice versa

   If there are no dropped or mangled messages, signal to the user "history verification successfull".

2.3.1 Device Loss

A typical scenario for a key change is device loss. The owner of the lost device loses access to his private key. We note that when this happens, in most cases the owner also loses access to his messages (because he can no longer decrypt them) and his key history.

Thus, if Bob lost his device, it is likely that Alice will have a much longer history for him then he has himself. Bob can only compare keys for the timespan after the device loss. While this verification is certainly less useful, it would enable Alice and Bob to detect of attacks in that time after the device lossj.

On the other hand, we can also envision users storing their history outside of their devices. The security requirements for such a backup are much lower than for backing up the private key. The backup only needs to be tamper proof, i.e., its integrity must be guaranteed :–: not its confidentiality. This is achievable even if the private key is lost. Users can verify the integrity of this backup even if they lose their private key. For example, Bob can cryptographically sign the key history using his current key. As long as Bob, and others, have access to Bob's public key, he can verify that the backup has not been tampered with.

An alternative is to permit that Bob recovers his history from the message/keydata list that he receives from Alice. Then, he could validate such information with other people in subsequent verifications. However, this method is vulnerable to collusion attacks in which Bob's keys are replaced in all of his peers, including Alice. It may also lead to other error cases that are much harder to investigate. We therefore discourage such an approach.

2.3.2 Keeping records of keys in messages

The history verification described above requires all e-mail apps (MUAs) to record,

- each e-mail address/key-fingerprint tuple it **ever** saw in an Autocrypt or an Autocrypt-Gossip header in incoming mails. This means not just the most recent one(s), but the full history.
- each emailaddr/key association it ever sent out in an Autocrypt or an Autocrypt Gossip header.

It needs to associate these data with the corresponding message-id.

State tracking suggested implementation

We suggest MUAs could maintain an outgoing and incoming "message-log" which keeps track of the information in all incoming and outgoing mails, respectively. A message with N recipients would cause N entries in both the sender's outgoing and each of the recipient's incoming message logs. Both incoming and outgoing message-logs would contain these attributes:

- `message-id`: The message-id of the e-mail
- `date`: the parsed Date header as inserted by the sending MUA
- `from-addr`: the sender's routable e-mail address part of the From header.
- `from-fingerprint`: the sender's key fingerprint of the sent Autocrypt key (NULL if no Autocrypt header was sent)
- `recipient-addr`: the routable e-mail address of a recipient

- `recipient-fingerprint`: the fingerprint of the key we sent or received in a gossip header (NULL if not Autocrypt-Gossip header was sent)

It is also possible to serialize the list of recipient addresses and fingerprints into a single value, which would result in only one entry in the sender's outgoing and each recipient's incoming message log. This implementation may be more efficient, but it is also less flexible in terms of how to share information.

2.3.3 Usability question of "sticky" encryption and key loss

Do we want to prevent dropping back to not encrypting or encrypting with a different key if a peer's autocrypt key state changes? Key change or drop back to cleartext is opportunistically accepted by the Autocrypt Level 1 key processing logic and eases communication in cases of device or key loss. The "setup-contact" also conveniently allows two peers who have no address of each other to establish contact. Ultimately, it depends on the guarantees a mail app wants to provide and how it represents cryptographic properties to the user.

2.4 Verifying keys through onion-queries

Up to this point this document has describe methods to securely add contacts, form groups, and verify history in an offline scenario where users can establish a second channel to carry out the verification. We now discuss how the use of Autocrypt headers can be used to support continuous key verification in an online setting.

A straightforward approach to ensure view consistency in a group is to have all members of the group continuously broadcasting their belief about other group member's keys. Unless they are fully isolated by the adversary (see Section for an analysis). This enables every member to cross check their beliefs about others and find inconsistencies that reveal an attack.

However, this is problematic from a privacy perspective. When Alice publishes her latest belief about others' keys she is implicitly revealing what is the last status she observed which in turn allows to infer when was the last time she had contact with them. If such contact happened outside of the group this is revealing information that would not be available had keys not been gossiped.

We now propose an alternative in which group members do not need to broadcast information in order to enable key verification. The solution builds on the observation that the best person to verify Alice's key is Alice herself. Thus, if Bob wants to verify her key, it suffices to be able to create a secure channel between Bob and Alice so that she can confirm his belief on her key.

However, Bob directly contacting Alice through the group channel reveals immediately that he is interested on verifying her key to the group members, which again raises privacy concerns. Instead, we propose that Bob relies on other members to rely the verifying message to Alice, similarly to a typical anonymous communication network.

The protocol works as follows:

1. Bob chooses n members of the group as relying parties to form the channel to Alice. For simplicity let us take $n = 2$ and assume these members are Charlie, key k_C, and David, with key k_D (both k_C and k_D being the current belief of Bob regarding Charlie and David's keys).
2. Bob encrypts a message of the form (`Bob_ID`, `Alice_ID` , k_A) with David and Charlie's keys in an onion encryption:

   E_{k_C} (`David_ID`, E_{k_D} (`Alice_ID`,(`Bob_ID`, `Alice_ID`, k_A))), where E_{k_*} indicates encrypted with key k_*

   In this message `Bob_ID` and `Alice_ID` are the identifiers, e.g., email addresses, that Alice and Bob use to identify each other. The message effectively encodes the question 'Bob asks: Alice, is your key k_A?'
3. Bob sends the message to Charlie, who decrypts the message to find that it has to be relayed to David.
4. David receives Charlie's message, decrypts and relays the message to Alice.
5. Alice receives the message and replies to Bob repeating steps 1 to 4 with other random n members and inverting the IDs in the message.

From a security perspective, i.e., in terms of resistance to adversaries, this process has the same security properties as the broadcasting. For the adversary to be able to intercept the queries he must MITM all the keys between Bob and others.

From a privacy perspective it improves over broadcasting in the sense that not everyone learns each other status of belief. Also, Charlie knows that Bob is trying a verification, but not of whom. However, David gets to learn that Bob is trying to verify Alice's key, thus his particular interest on her.

This problem can be solved in two ways:

1. All members of the group check each other continuously so as to provide plausible deniability regarding real checks.
2. Bob protects the message using secret sharing so that only Alice can see the content once all shares are received. Instead of sending (`Bob_ID`, `Alice_ID` , k_A) directly, Bob splits it into t shares. Each of this shares is sent to Alice through a *distinct* channel. This means that Bob needs toe create t channels, as in step 1.

   When Alice receives the t shares she can recover the message and respond to Bob in the same way. In this version of the protocol, David (or any of the last hops before Alice) only learns that someone is verifying Alice, but not whom, i.e., Bob's privacy is protected.

2.4.1 Open Questions about onion online verification

An open question is how to choose contacts to rely onion verification messages. This choice should not reveal new information about users' relationships nor the current groups where they

belong. Thus, the most convenient is to always choose members of the same group. Other selection strategies need to be analyzed with respect to their privacy properties.

The other point to be discussed is bandwidth. Having everyone publishing their status implies N*(N-1) messages. The proposed solution employs 2*N*n*t messages. For small groups the traffic can be higher. Thus, there is a tradeoff privacy vs. overhead.

3 Key consistency with ClaimChains

In this section we show how ClaimChains, a data structure that can be used to store users' key history in a secure and privacy-preserving way, can be used to support keyhistory verification; and can also be used to identify which contacts are best suited to perform in-person key verifications.

We first provide a brief introduction to the ClaimChains structure and its properties. Then, we describe a concrete usage of ClaimChains in the Autocrypt context.

3.1 High level overview of the ClaimChain design

ClaimChains store *claims* that users make about their keys and their view of others' keys. The chain is self-authenticating and encrypted. Cryptographic access control is implemented via capabilities. In our design, the chains are stored as linked blocks with a publicly accessible block storage service in a privacy-preserving way.

Claims come in two forms: self-claims, in which a user shares information about her own key material, and cross-references, in which a user vouches for the key of a contact.

A user may have one or multiple such ClaimChains, for example, associated with multiple devices or multiple pseudonyms.

ClaimChains provide the following properties:

- **Privacy of the claim it stores,** only authorized users can access the key material and cross-references being distributed.
- **Privacy of the user's social graph,** nor providers nor unauthorized users can learn whose contacts a user has referenced in her ClaimChain.

Additionally ClaimCains are designed to prevent *equivocation*. That is, given Alices ClaimChain, every other user must have the same view of the cross-references. In other words, it cannot be that Carol and Donald observe different versions of Bob's key. If such equivocation were possible, it would hinder the ability to resolve correct public keys.

3.1.1 The ClaimChain Design

ClaimChains represent repositories of claims that users make about themselves or other users. To account for user beliefs evolving over time, ClaimChains are implemented as cryptographic hash chains of blocks. Each block of a ClaimChain includes all claims that its owner endorses at the point in time when the block is generated, and all data needed to authenticate the chain. In order to optimize space, it is possible to only put commitments to claims in the block, and offload the claims themselves onto a separate data structure.

Other than containing claims, each block in the chain contains enough information to authenticate past blocks as being part of the chain, as well as validate future blocks as being valid updates.

Thus, a user with access to a chain block that they believe provides correct information may both audit past states of the chain, and authenticate the validity of newer blocks. In particular, a user with access to the *head* of the chain can validate the full chain.

We consider that a user stores three types of information in a ClaimChain:

- **Self-claims.** Most importantly these include cryptographic encryption keys. There may also be other claims about the user herself such as identity information (screen name, real name, email or chat identifiers) or other cryptographic material needed for particular applications, like verification keys to support digital signatures. Claims about user's own data are initially self-asserted, and gain credibility by being cross-referenced in chains of other users.
- **Cross-claims.** The primary claim about another user is endorsing other user's ClaimChain as being authoritative, i.e. indicate the belief that the key material found in the self-claims of those chains is correct.
- **Cryptographic metadata.** ClaimChains must contain enough information to authenticate all past states, as well as future updates of the repository. For this purpose they include digital signatures and corresponding signing public keys.

In order to enable efficient operations without the need for another party to have full visibility of all claims in the chain, ClaimChains also have cryptographic links to past states. Furthermore, blocks include roots of high-integrity data structures that enable fast proofs of inclusion of a claim in the ClaimChain.

Any of the claims can be public(readable by anyone), or private. The readability of private claims on a chain is enforced using a cryptographic access control mechanism based on capabilities. Only users that are provided with a capability for reading a particular cross-reference in a ClaimChain can read such claim, or even learn about its existence.

Other material needed for ensuring privacy and non-equivocation is also included, as described in detail at https://claimchain.github.io .

3.2 Use and architecture

This section discusses how ClaimChains can be integrated into Autocrypt. It considers that:

- ClaimChains themselves are retrieved and uploaded from an online storage whenever a message is sent or received,
- ClaimChain heads are transferred using email headers.

This version is being implemented at https://github.com/nextleap-project/muacryptcc .

3.2.1 Inclusion in Messages

When Autocrypt gossip includes keys of other users in an email claims about these keys are included in the senders chain. The email will reference the senders chain as follows:

The Autocrypt and gossip headers are the same as usual. In addition we include a single header that is used to transmit the sender head imprint (root hash of our latest CC block) in the encrypted and signed part of the message:

```
GossipClaims: <head imprint of my claim chain>
```

Once a header is available, the corresponding ClaimChain block(s) can be retrieved from the block storage service. After retrieving the chain the recipients can verify that the other recipients keys are properly included in the chain.

The block also contains pointers to previous blocks such that the chain can be efficiently traversed.

3.2.2 Mitigating Equivocation in different blocks

The easiest way to circumvent the non-equivocation property is to send different blocks to two different parties.

We work around this by proving to our peers that we did not equivocate in any of the blocks.

The person who can best confirm the data in a block is the owner of the respective key.

3.2.3 Proofs of inclusion

Proofs of inclusion allow verifying the inclusion of claims in the chain without retrieving the entire block.

The ClaimChain design suggests to include proofs of inclusion for the gossiped keys in the headers. This way the inclusion in the given block could be verified offline.

However in order to prevent equivocation all blocks since the last one we know need to be checked. Therefore we would have to include proofs of inclusion for all recipients and for all blocks since they last saw the chain. This in turn would require tracking the state each peer last saw of our own chain.

We decided against adding the complexity involved. Instead we require users to be online to verify the inclusion of their own keys in peers chains and the overall consistency of their peers claims.

This fits nicely with the recommendation guidance workflow described below.

3.2.4 Constructing New Blocks

The absence of a claim can not be distinguished from the lack of a capability for that claim. Therefore, to prove that a ClaimChain is not equivocating about keys gossiped in the past they need to include, in every block, claims corresponding to those keys, and grant access to all peers with whom the key was shared in the past.

When constructing a new block we start by including all claims about keys present in the last block, and their corresponding capabilities.

In addition the client will include claims with the fingerprints of new gossiped keys. For peers that also use ClaimChain the client will include a cross-reference, i.e., the root hash of the latest block they saw from that peer in the claim.

Then, if they did not exist already, the client will grant capabilities to the recipients for the claims concerning those recipients. In other words, it will provide the recipients with enough information to learn each other keys and ClaimChain heads.

Note that due to the privacy preserving nature of ClaimChain these keys will not be revealed to anyone else even if the block data is publically accessible.

3.3 Evaluating ClaimChains to guide verification

Verifying contacts requires meeting in person, or relying on another trusted channel. We aim at providing users with means to identify which contacts are the most relevant to validate in order to maintain the security of their communication.

The first in-person verification is particularly important. Getting a good first verified contact prevents full isolation of the user, since at that point it is not possible anymore to perform MITM attacks on all of her connections.

Due to the small world phenomenon in social networks few verifications per user will already lead to a large cluster of verified contacts in the social graph. In this scenario any MITM attack will lead to inconsistencies observed by both the attacked parties and their neighbours. We quantify the likelihood of an attack in *Attack Scenarios* (page 28).

To detect inconsistencies clients can compare their own ClaimChains with those of peers. Inconsistencies appear as claims by one peer about another peer's key material that differ from ones own observation.

Given inconsistency of a key it is not possible to identify unequivocally which connection is under attack:

- It may be the connection between other peers that leads them to see MITM keys for each other, while the owner is actually observing the actual ones.
- It may be that the owner is seeing MITM keys for one of them, while the other one is claiming the correct key.

Verifying one of the contacts for whom an inconsistency has been detected will allow determining whether that particular connection is under attack. Therefore we suggest that the recommendation regarding the verification of contacts is based on the number of inconsistencies observed.

3.3.1 Split world view attacks

Note, however, that the fact that peers' claims are consistent does not imply that no attack is taking place. It only means that to get to this situation an attacker has to split the social graph into groups with consistent ideas about their peers keys. This is only possible if there are no verified connections between the different groups. It also requires mitm attacks on more connections possibly involving different providers. Therefore checking consistency makes the attack both harder and easier to detect.

In the absence of inconsistencies we would therefore like to guide the user towards verifying contacts they have no (multi-hop) verified connection to. But since we want to preserve the privacy of who verified whom we cannot detect this property. The best guidance we can offer is to verify users who we do not share a verified group with yet.

3.3.2 Inconsistencies between other peoples chains

In addition to checking consistency with the own chain the clients could also compare claims across the ClaimChains of other people. However, inconsistencies between the chains of others are a lot harder to investigate. Therefore their use for guiding the user is very limited. Effectively the knowledge about conflicts between other peoples chains is not actionable for the user. They could verify with one of their peers - but even that would not lead to conclusive evidence.

In addition our implementation stores claims about all keys in active use in its own claimchain. Therefore if the user communicates with the person in question at least one of the conflicting keys of peers will conflict with our own recorded key. We refrain from asking the user to verify people they do not communicate with.

3.3.3 Problems noticed

- complex to specify interoperable wire format of ClaimChains and all of the involved cryptographic algorithms
- Autocrypt-gossip + DKIM already make it hard for providers to equivocate. CC don't add that much (especially in relation to the complexity they introduce)
- lack of underlying implementation for different languages
- Maybe semi-centralized online storage access (we can postpone storage updates to the time we actually send mail)

4 Using Autocrypt key gossip to guide key verification

Autocrypt Level 1 introduces key gossip[7] where a sender adds `Autocrypt-Gossip` headers to the encrypted part of a multi-recipient message. This was introduced to ensure users are able to reply encrypted. Because according to the Autocrypt specification encrypted message parts are always signed, recipients may interpret the gossip keys as a form of third-party verification.

In *gossip-attack* (page 28) we look at how MUAs can check key consistency with respect to particular attacks. MUAs can flag possible machine-in-the-middle (mitm) attacks on one of the direct connections which in turn can be used for helping users with prioritizing *History-verification protocol* (page 17) with those peers. To mitigate, attackers may intercept multiple connections to split the recipients into mostly isolated groups. However, the need to attack multiple connections at once increases the chance of detecting the attack by even a small amount of Out-of-Band key verifications.

The approaches described here are applicable to other asymmetric encryption schemes with multi recipient messages. They are independent of the key distribution mechanism - wether it is in-band such as in Autocrypt or based on a keyserver like architecture such as in Signal.

4.1 Attack Scenarios

4.1.1 Attacking group communication on a single connection

Fig. 4: Targetted attack on a single connection

The attacker intercepts the initial message from Alice to Bob (1) and replaces Alices key `a` with a mitm key `a'` (2). When Bob replies (3) the attacker decrypts the message, replaces Bobs key `b` with `b'`, encrypts the message to `a` and passes it on to Alice (4).

Both Bob and Alice also communicate with Claire (5,6,7,8). Even if the attacker chooses to not attack this communication the attack on a single connection poses a significant risk for group communication amongst the three.

Since each group message goes out to everyone in the group the attacker can read the content of all messages sent by Alice or Bob. Even worse ... it's a common habit in a number of messaging systems to include quoted text from previous messages. So despite only targetting two participants the attack can provide access to a large part of the groups conversation.

Therefore participants need to worry about the correctness of the encryption keys they use but also of those of everyone else in the group.

[7] https://autocrypt.org/level1.html#key-gossip

4.1.2 Detecting mitm through gossip inconsistencies

Some cryptographic systems such as OpenPGP leak the keys used for other recipients and schemes like Autocrypt even include the keys. This allows checking them for inconsistencies to improve the confidence in the confidentiality of group conversation.

Fig. 5: Detecting mitm through gossip inconsistencies

In the scenario outlined above Alice knows about three keys (`a`, `b'`, `c`). Sending a message to both Bob and Clair she signs the message with her own key and includes the other two as gossip keys `a[b',c]`. The message is intercepted (1) and Bob receives one signed with `a'` and including the keys `b` and `c` (2). Claire receives the original message (3) and since it was signed with `a` it cannot be altered. C's client can now detect that A is using a different key for B (4). This may have been caused by a key update due to device loss. However if B responds to the message (5,6,7) , C learns that B also uses a different key for A (8). At this point C's client can suggest to verify fingerprints with either A or B. In addition a reply by C (9, 10) will provide A and B with keys of each other through an independent signed and encrypted channel. Therefore checking gossip keys poses a significant risk for detection for the attacker.

4.1.3 Attacks with split world views

In order to prevent detection through inconsistencies an attacker may choose to try and attack in a way that leads to consistent world views for everyone involved. If the attacker in the example above also attacked the key exchange between A and C and replaced the gossip keys accordingly here's what everyone would see:

```
A: a , b', c'
B: a', b , c
C: a', b , c
```

Only B and C have been able to establish a secure communication channel. But from their point of view the key for A is a' consistently. Therefore there is no reason for them to be suspicious.

Note however that the provider had to attack two key exchanges. This increases the risk of being detected through OOB-verification.

4.2 Probability of detecting an attack through out of band verification

Attacks on key exchange to carry out mitm attacks that replace everyones keys would be detected by the first out-of-band verification and the detection could easily be reproduced by others.

However if the attack was carried out on only a small part of all connections the likelyhood of detection would be far lower and error messages could easily be attributed to software errors or

other quirks. So even an attacker with little knowledge about the population they are attacking can learn a significant part of the group communication without risking detection.

In this section we will discuss the likelyhood of detecting mitm attacks on randomly selected members of a group. This probabilistic discussion assumes the likelyhood of a member being attacked as uniform and independent of the likelyhood of out-of-band verification. It therefore serves as a model of randomly spread broad scale attacks rather than targetted attacks.

4.2.1 Calculating the likelyhood of detection

A group with n members has $c = n \times \frac{n-1}{2}$ connections.

Let's consider an attack on a connections. This leaves $g = c - a$ good connections. The probability of the attack not being detected with 1 key verification therefore is $\frac{g}{c}$.

If the attack remains undetected c-1 unverified connections amongst which (g-1) are good remain. So the probability of the attack going unnoticed in v verification attempts is:

$$\frac{g}{c} \times \frac{g-1}{c-1} \ldots \times \frac{g-(v-1)}{c-(v-1)} = \frac{g(g-1)\ldots(g-(v-1))}{c(c-1)\ldots(c-(v-1))} = \frac{\frac{g!}{(g-v)!}}{\frac{c!}{(c-v)!}} = \frac{g!(c-v)!}{c!(g-v)!}$$

4.2.2 Single Attack

As said above without checking gossip an attacker can access a relevant part of the group conversation and all direct messages between two people by attacking their connection and nothing else.

In order to detect the attack key verification needs to be performed on the right connection. In a group of 3 users there are 3 direct connections. Therefor the chance of a single key verificatoin for detecting the attack is $\frac{1}{3}$. In a group of 10 the chances are even slimmer: *frac{1}{45} approx 2%*

4.2.3 Isolation attack

Isolating a user in a group of n people requires (n-1) interceptions. This is the smallest attack possible that still provides consistent world views for all group members. Even a single verification will detect an isolation attack with a probability > 20% in groups smaller than 10 people and > 10% in groups smaller than 20 people.

Isolation attacks can be detected in all cases if every participant performs at least 1 OOB-verification.

4.2.4 Isolating pairs

If each participant OOB-verifies at least one other key isolation attacks can be ruled out. The next least invasive attack would be trying to isolate pairs from the rest of the group. However this

requires more interceptions and even 1 verification on average per user leads to a chance > 88% for detecting an attack on a random pair of users.

4.2.5 Targeted isolation

The probabilities listed in the table assume that the attacker has no information about the likelyhood of out of band verification between the users. If a group is known to require a single key verification per person and two members of the group are socially or geographically isolated chances are they will verify each others fingerprints and are less likely to verify fingerprints with anyone else. Including such information can significantly reduce the risk for an attacker.

5 Using DKIM signature checks to guide key verification

With DomainKeys Identified Mail (DKIM)[8], a mail transfer agent (MTA) signals to other MTAs that a particular message passed through one of its machines. In particular, a MTA signs outoing mail from their users with a public key that is stored with DNS, the internet domain name system. The MTA adds a `DKIM-Signature` header which is then verified by the next MTA which in turns may add an Authentication-Results header[9]. After one or more MTAs have seen and potentially DKIM-signed the message, it finally arrives at Mail User Agents (MUAs). MUAs then can not reliably verify all DKIM-signatures because the intermediate MTAs may have mangled the original message, a common practise with mailing lists and virus-checker software.

In *DKIM Signatures on Autocrypt Headers* (page 32) and following we discuss how DKIM-signatures can help protect the Autocrypt key material from tampering between the senders MTA and the recipients MUA.

5.1 DKIM Signatures on Autocrypt Headers

Fig. 6: Sequence diagram of Autocrypt key exchange with DKIM Signatures

Alice sends a mail to Bob including an Autocrypt header with her key(a). First, Alice's Provider authenticates Alice, and upon receiving her message (1), it adds a DKIM signature header and then passes it on to Bobs provider (2). When Bob's provider receives the message it retrieves the public DKIM key from Alice's provider (3,4) and verifies Alice's provider DKIM signature (5).

This is the default DKIM procedure and serves primarily to detect and prevent spam email. If the DKIM signature matches (and other spam tests pass) Bob's provider relays the message to Bob (6).

In the current established practice Bob's MUA will simply present the message to Bob without any further verification. This means that Bob's provider is in a position to modify the message before it is presented to Alice. This can be avoided if Bob's MUA also retrieves the DKIM key (7,8) and verifies the signature (9, making sure that the headers and content have not been altered after leaving the Alice's provider. In other words, a valid DKIM signature on the mail headers, including the Autocrypt header, indicates that the recipient's provider has not altered the key included in the header.

It must be noted that since some providers do not use DKIM signatures at all, a missing signature by itself does not indicate a MITM attack. Also, some providers alter incoming mails to attach mail headers or add footers to the message body. Therefore even a broken signature can have a number of causes.

[8] https://dkimorg
[9] https://en.wikipedia.org/wiki/Email_authentication#Authentication-Results

The DKIM header includes a field `bh` with the hash of the email body that was used to calculate the full signature. If the DKIM signature is broken it may still be possible to verify the Autocrypt header based on the body hash and the signed headers.

5.2 Device loss and MITM attacks

Autocrypt specifies to happily accept new keys send in Autocrypt headers even if a different key was received before. This is meant to prevent unreadable mail, but also offers a larger attack surface for MITM attacks.

The Autocrypt spec explicitely states that it does not provide protection against active attacks. However combined with DKIM signatures at least a basic level of protection can be achieved:

A new key distributed in a mail header with a valid DKIM signature signals that the key was not altered after the mail left the sender's provider. Yet, the following threats remain:

- the sender's device was compromised
- the sender's email account was compromised
- the transport layer encryption between the sender and their provider was broken
- the sender's provider is malicious
- the sender's provider was compromised

This attack vector is shared by any other key distribution scheme that rely on the provider to certify or distribute the user's keys.

5.2.1 One malicious provider out of two

In order to carry out a successful transparent MITM attack on a conversation the attacker needs to replace both parties keys and intercept all mails. While it's easy for either one of the providers to intercept all emails replacing the keys in the headers and the signatures in the body will lead to broken DKIM signatures in one direction.

5.2.2 Same provider or two malicious providers

If both providers cooperate on the attack or both users use the same provider it's easy for the attacker to replace the keys and pgp signatures on the mails before DKIM signing them. However, with their DKIM-signatures they would have signed Autocrypt headers that were never sent by the users's MUAs.

5.2.3 Key updates in suspicious mails

If a MUA has seen an Autocrypt header with a valid DKIM signature from the sender before and receives a new key in a mail without a signature or with a broken signature that may indicate a MITM attack.

5.3 Open Questions

5.3.1 Reliability of DKIM signatures

Key update notifications suffer from a high number of false positives. Most of the time the key holder just lost their device and reset the key. How likely is it that for a given sender and recipient DKIM signatures that used to be valid when receiving the emails stop being valid? How likely is this to occure with the introduction of a new key? Intuitively both events occuring at the same time seems highly unlikely. However an attacker could also first start breaking DKIM signatures and insert a new key after some mails. In order to estimate the usefulness of this approach more experiences with MUA side validation of DKIM signatures would be helpful.

5.3.2 Provider support

In December 2017 the provider posteo.de announced that they will DKIM sign Autocrypt headers of outgoing mail.

What can providers do?

- DKIM-sign Autocrypt headers in outgoing mails
- preserve DKIM signed headers in incoming mails
- add an `Authentication-Results` header which indicates success in DKIM validation.

Maybe they can indicate both these properties in a way that can be checked by the recipients MUA?