NeXt generation Techno-social Legal Encryption Access and Privacy nextleap.eu
Grant No. 688722 Project Started 2016-01-01. Duration 36 months

DELIVERABLE D5.3

Validated Modules for Federated Identity and Privacy-Preserving End-to-End Encrypted Messaging

Holger Krekel (merlinux)
Azul (merlinux)
Björn Petersen (merlinux)

Beneficiaries:	merlinux, IMDEA, UCL
Description:	Based on feedback and user-experience "lean design" rounds prototypes with the selected user-bases will be refactored and improved to the point of being ready for integration with real-life projects.

Version:	**1.0**
Nature:	**Report (R)**
Dissemination level:	**Public (P)**
Pages:	**24**
Date:	**2018-12-31**
Reviewers:	**Carmela Troncoso (EPFL), Ksenia Ermoshina (CNRS)**

Abstract: In this report, we describe our efforts in the second reporting period regarding implementing decentralized protocols for NEXTLEAP federated identity and end-to-end encrypted messaging protocols. The report discusses the software releases regarding ClaimChain/Autocrypt integration architecture and new key verification protocols for messaging.

1. Overview and introduction

In this deliverable we summarize advances in WP5 regarding integration of protocols developed within other Work Packages. These protocols also have inputs from partners external to the project gathered in meetings and public email exchanges. It summarizes and puts the NEXTLEAP protocols into a coherent research and implementation context of secure, decentralised e2e-encrypted and privacy-preserving messaging on top of the e-mail protocol server deployment.

We have released several software modules and packages to ensure a coherent integration between ClaimChain (D2.2) and Autocrypt functionality which we detail in Section 2. In Section 3 we describe our lab implementation releases of the new NEXTLEAP key verification protocols (D2.3) for privacy-preserving and end-to-end encryption.

These protocols were developed jointly with our partners from WP2 and WP4, resulting in a collaborative technical report on countering active attacks against Autocrypt. The report is based on NEXTLEAP work meetings at EPFL in Lausanne, in December 2017, and January 2018. The first version of the report was published end May 2018. It was since updated to include feedback from other researchers such as Bryan Ford (EPFL) and Michael Rogers (Briar Project).

Besides integration of the core security protocols underlying Autocrypt, we continued discussions with different encrypted email projects in order to improve the design of our user interface (UI) flows.

1.1. Summary of developments since D5.3a

We updated the technical report with feedback from other researchers, internal and external to the project. In particular we now prevent replay attacks and designed verified group and history verification to be more closely based on contact verification. We also refined and documented the interaction of encryption with verified and opportunistic Autocrypt keys. In collaboration with Carmela Troncoso and Wouter Lueks (EPFL) we improved readability and clarity in the entire document. We addressed feedback from various researchers from the messaging mailing list and released 0.10.0 of the public NEXTLEAP countermitm document[1].

[1] https://countermitm.readthedocs.io

We presented both the report and the inclusion of the protection against active attacks in delta.chat at the OpenPGP Summit in October 2018 in Brussels.

1.1.1 Validated modules for federated identity

- We implemented the federated block storage (cchttpserver) and synchronization commands in muacryptcc to complete ClaimChain/Autocrypt integration. See “Use online storage to exchange claims (0.9.0)” for details about the release, Section 2.2.
- Both muacrypt and DeltaChat implementations have been further tested with K-9 Mail on Android, Enigmail/Thunderbird and incompatibilities were fixed.
- Formal modelling in D4.1 already informed the development of ClaimChains. Therefore its results were already included in the design. The Implementation of ClaimChains in the context of Autocrypt revealed that care needs to be taken to prevent equivocation between different blocks. The latest ClaimChain design takes these considerations into account in a privacy preserving manner. This improvement in the design is reflected in D4.4.
- The simulations in D4.2 show that claimchains are scalable with low overhead to the message size and small block sizes that can easily be distributed with the federated block storage.

1.1.2 Validated modules for privacy preserving e2e encryption

- We integrated muacrypt with the mutt email application to enable Autocrypt based encryption and the use of claimchains via muacryptcc.
- The “verified contact setup” and “verified group” protocol implementations have been tested and successfully validated in two user-testing sessions in Kyiv, Ukraine, on October 30 and October 31. Subsequent refinements and key-history-verification work will be funded through OpenTechnologyFund in 2019 which accepted the “Delta Chat Usability and Robustness” proposal.
- The public muacrypt-0.9.0 release is now ready for integration into projects like mailman (mailing list manager software) and LEAP (Leap encryption access project).
- Formal modelling of Autocrypt, and the verified contact setup protocols revealed shortcomings in the underlying OpenPGP standard (D4.3). In particular an AEAD based design would prevent these weaknesses. The standard is evolving slowly with RFC4880bis. We got involved with discussions around this in the context of the OpenPGP summit. Implementations still have to wait for a consensus to be reached around the definition of the standard, though.
- The Email ecosystem is known to be scalable. Therefore D4.4 focused on preventing traffic analysis. The proposed measures complement Autocrypt and ClaimChain by preventing privacy leaks through traffic patterns.

2. ClaimChain / Autocrypt integration modules

This section discusses our implementation of a ClaimChain system to work alongside Autocrypt. It uses email headers to transfer references to the claimchains of the sender and recipients. The chains themselves are uploaded and retrieved from an online storage at message delivery and retrieval times.

2.1. ClaimChain and Autocrypt integration architecture

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

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 ClaimChains are designed to prevent equivocation. That is, given Alice's 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.

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 .

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 .

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 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.

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.

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 for each peer the state they last saw of our own chain.

In addition we need to consider multi device scenarios in which both the sender and recipient might be using different devices: We have limited information about which device of our peer received which messages and we cannot predict which of them will receive the message in composition.

We decided against adding the complexity involved. Instead we require users to be online when verifying 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.

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 publicly accessible.

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.

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 an 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.

Split worldview 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 a consistent view 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.

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.

2.2. Software module releases

The current ongoing ClaimChain/Autocrypt implementation work takes place in the following locations:

- https://github.com/nextleap-project/muacryptcc for the ClaimChain/Autocrypt integration with muacrypt
- https://github.com/hpk42/muacrypt for the base autocrypt and command line tool implementations (also used for bot@autocrypt.org)
- https://github.com/claimchain/claimchain-core for the core ClaimChain implementation
- https://github.com/gdanezis/hippiehug for the merkle tree implementation for ClaimChain.

All modules have had multiple iterative releases which we list in the following sections along with change details. They are released on github and follow good practices in terms of testing and documentation. Release versions are given in brackets for each release listed below.

MuacryptCC

Use online storage to exchange claims (0.9.0)

This release introduces the cc-send command. It will upload the local claim chain to a remote cchttpserver.

FileStore now takes the url of a remote cchttpserver as an argument. When claims are not available locally it will look them up remotely. This way it acts as a transparent cache when reading peers claim chains.

This release provides the following:

- use cc_account.upload() in the cc-send command
- explicitly call cc.upload() to upload new blocks
- reuse the existing plugin, when initialization happens twice unregistering the old and registering a new plugin might cause problems if the old CC account is still used somewhere.
- cc-status: print some more details
- filestore: recv missing data from remote. This way we can easily integrate it with claimchain. For other peoples chains the store basically acts as a local cache.
- enable FileStore to sync to a remote cchttpserver
- use devpi-index for getting latest "muacrypt"
- use muacrypt's command line structure where accounts are always specified via "-a ACCOUNTNAME" and default to "default"
- rename cc-sync to cc-send and make it accept a URL
- fix str/bytes issues

Fix packaging issues (0.8.1 and 0.8.2)

This release fixes some packaging issues and depends on the separate "ClaimChain" and "hippiehug" module releases. It also completes the shift from the prior autocrypt naming to the new muacrypt name.

Initial public release - asserting consistency (0.8.0)

This is the first release that can be used to verify consistency of the keys observed. It will persist a log about them and raise assertion errors in case of inconsistent keys.

MuacryptCC is established as a plugin to Muacrypt. It makes use of hooks into muacrypt.

This release does not yet allow retrieving chains from peers as it relies on local files as a chain store.

This release provides the following:

- use own claimchain to store info about peers including the public dh key for their claimchain.
- implement CCAccount to handle all ClaimChain related operations. It abstracts away the detailed calls to add claims and capabilities. Instead it operates on concepts like peers and chains. It defines the concrete format for claims.
- add initial ClaimChain subcommands to muacrypt
- make use of peer info to add capabilities for peers with ClaimChains.

- register peers and store info about them in our own ClaimChain.
- include head imprints and store urls for ClaimChains of peers if available
- add claims according to gossip present in the outgoing mails.
- unit tests for the CCAccount module and integration test for the use as a Muacrypt module
- establish internal API for storing and retrieving claims. This API can also be used to read claims from other peoples chains if the required capabilities are present.
- build test system that includes muacrypt and makes use of it's hook system
- Make use of muacrypt hooks to learn about messages received and inject ClaimChain headers into outgoing mails.

We also provided error reports, failing tests and various fixes to claimchain-core and rousseau-chain. ClaimChain version 0.3.0 incorporates all fixes.

We rely on hooks provided by Muacrypt version 0.8.0. MuacryptCC will not be able to register commands with previous versions.

Muacrypt

Mailclient (Mutt) integration (0.9.0)

- support and document a viable mutt/muacrypt integration
- all subcommands which take an account name now do it through the "-a" or "--account" option.
- add "muacrypt import-public-key" subcommand to integrate a key with a specified prefer-encrypt setting and e-mail address.
- add "scandir-incoming" subcommand to scan maildirs for incoming mail and Autocrypt headers.
- add "peerstate EMAILADR" command which shows Autocrypt and key state for a given peer.
- renamed "test-email" to "find-account" subcommand as it is about finding the account for a particular (own) e-mail address.
- make muacrypt fail by default in process-outgoing/sendmail if no muacrypt account could be determined for an outgoing mail
- fix test suite with --no-test-cache run
- disable warnings for pytest_localserver's smtp support
- add muacrypt version to pytest report header
- refine tests for process-incoming and autocrypt timestamps

- use stable serializers/unserializers from the cross-py2/py3 execnet package

Fix packaging and test failure issues (0.8.2)

- fix project description

Allow unicode ‘To’ addresses (0.8.1)

- fixed changelog
- allow unicode To addresses in process-sendmail/outgoing

Introduce plugin architecture and Autocrypt gossip (0.8.0)

- introduce plugin architecture with hooks on incoming/outgoing mail and for adding new subcommands.
- Release version and upload to pypi
- reply to multiple CC'ed recipients with the bot so we can test gossip.
- add Autocrypt-Gossip headers to mails with multiple recipients.
- parse gossip headers with the bot.
- refine recommendations and add command line call
- add way to add subcommands from a plugin
- moved repo to hpk42/py-autocrypt and refined entry pages to link to new IRC channel and mailing list and describe the aims.
- removed "init" subcommand. you can now directly use "add-account".
- completely revamped internal storage to use append-only logs. all state changes (and in particular Autocrypt header processing) is tracked in immutable entries.
- with gpg2 we now internally use a hardcoded passphrase to avoid problems with gpg-2.1.11 on ubuntu 16.04 which does not seem to allow no-passphrase operations very well.
- #22 introduce account.encrypt_mime and account.decrypt_mime API (not yet exposed to cmdline).
- make tests work against gpg 2.0.21, gpg-2.1.11 (and likely higher versions but those are hard to custom-build on ubuntu or older debian machines)
- introduce decrypt/encrypt support for the bot and implement the autocrypt Level 1 recommendation algorithm for determining if encryption should happen or not.

Level 1 compatibility (0.7.0)

- Rename package from “py-autocrypt” to “muacrypt”
- adapt Autocrypt header attribute names, parsing and processing to new Level 1 spec
- add "pgpy" backend but do not activate it yet because current pgpy versions are not compatible enough to gpg's crypto.
- change "sendmail" and "process-outgoing" commands to not add autocrypt headers if no account can be determined for a mail.
- add first version of "ClaimChain" code which py-autocrypt is to use for its internal key management. Claimchains are an append-only log of claims about cryptographic key material.

ClaimChain

releases to pypi (0.3.0 and 0.2.6)

- Upload wheels
- Fix a unicode / string issues
- Move usage warning above usage section
- Rework README to include usage instructions
- Test based on pypi packages rather than requirements and git
- Port test documenting issue discovered in muacryptcc

Prepare for public release (0.2.5)

- Add installation instructions
- Cleanup

Minor tweaks (0.2.4)

- Update Setup.py
- Add Zenodo badge

Improve documentation and update (0.2.3)

- Update dev requirements

- Update README and doc index
- Rename usage doc
- Add to readthedocs
- Add Manifest

Update hippiehug, add documentation (0.2.2)

- Add Documentation
- Clean up, remove unnecessary packages
- Update hippiehug

Fix internal encoding issue (0.2.1)

- No interface changes
- Fix FFI.string issue

Use ClaimChain organization in Github (0.2.0)

- Updated library dependencies and dropped redundant ones
- Added docs

Read own claims and export private keys (0.1.3)

- Owner retrieves claim using View without capability to herself
- Move LocalParams.get_default() in state.View constructor
- Fix example code in README
- Ensure nonce is binary in core.encode_claim()
- private export of identity
- Fix packaging

Hippiehug

Bugfixes based on findings in muacryptcc (0.1.3)

- include regular tests in tox run
- fix unicode/bytes issues with tests

- add manifest
- make sure to include values in hash calculation
- sort dicts before calculating hashes
- failing test for usage of hashes when copying Block

Improve `Block` constructor resilience plus minor fixes (0.1.2)

- make `Block` constructor more resilient
- fix various issues and use a "rstore" fixture
- fix py35 compat for speedtest files
- use pytest.importoskip('redis') for skipping redis tests

3. Lab releases of new “key verification” protocols

We integrated the new NEXTLEAP verification protocols[2] in lab releases for the Delta.Chat[3] messenger. Delta.Chat is a unique effort in the messaging ecosystem that offers a modern "chatty" Telegram-based UI combined with an e-mail backend, currently released on Android. End-to-end encryption is achieved using Autocrypt, which makes use of standard OpenPGP-based cryptographic operations. We choose Delta.Chat as our instant-messaging target because it naturally integrates with our focus on utilizing and improving the security of the e-mail ecosystem. With the May 2018 “Labs” releases of Delta.Chat we have implemented the basic “Setup contact” and “Verified group” NEXTLEAP protocols. These protocols protect users against active provider attacks while offering more convenient user interfaces for establishing contact or group membership. Early user testing on May 17th 2018 at Hackarnaval in Paris yielded positive feedback and bug reports which were subsequently fixed.

In this section we first provide some background about why Delta.Chat is an interesting implementation integration target for synchronous messaging. We describe details and screenshots of the new key verification features and finally list the releases which incorporated work done during WP5.

Delta.Chat: a decentralized e2e-encrypted messaging solution

While existing instant messaging solutions offer high grade, custom-built end-to-end encryption they also have the following problems:

[2] http://countermitm.readthedocs.io/en/latest/new.html
[3] https://delta.chat

- dangers of tying messaging identity to mobile phone numbers

- privacy concerns around centralized databases of metadata

- vulnerability of centralized servers to censorship and attack

- lack of reach and of interoperability, resulting in messaging silos

- lack of options for groups to self-host their secure messaging environment

By using the traditional federated e-mail infrastructure, some of these

problems can be mitigated but new problems arise:

- lack of usable and pervasive e-mail encryption to defend against surveillance

- lack of modern “Chat” UIs on top of the e-mail federated ecosystem

Delta.Chat offers a complementary secure messaging app that proposes to address and mitigate both sets of real-life problems. The independently funded DeltaChat Needfinding report [4] from December 2018, written by Xenia Ermoshina and Vadym Hudyma (both not NEXTLEAP at the time of writing) goes into more details of “Secure Messaging” needs and will further inform development beyond the NEXTLEAP project period.

3.1. Delta.Chat key verification architecture

To withstand network adversaries, key verification between peers is necessary to establish trustable e2e-encrypted group communication. Note that **key consistency** schemes do not remove the need to perform key verification. It is possible to have a group of peers which each see consistent email-addr/key bindings from each other, but a peer is consistently isolated by a machine-in-the-middle attack from a network adversary. It follows that each peer needs to verify with at least one other peer to assure that there is no isolation attack. See the countermitm site for more in-depth discussion on the verification protocols[5].

The key verification protocols in Delta.Chat use regular Autocrypt headers for key transportation but treat *verified* keys separately from normal, opportunistic keys. Verification starts off with a bootstrap QR code which is shown by one user and scanned by another. After a successful QR out-of-band-verification, the key matching the verified fingerprint is copied to a separate column in the peerstate-table. This verified key can then be used to create verified groups.

[4] https://delta.chat/assets/blog/dcneedfindingreport.pdf (independently funded)

[5] http://countermitm.readthedocs.io/en/latest/new.html#securing-communications-against-network-adversaries

Setup contact protocol

The “setup contact” protocol uses email messages to perform the handshake from the NEXTLEAP countermitm-paper[6]. For starting the handshake, the inviting device (Alice) generates a QR code that follows the OPENPGP4FPR-scheme (whitelisted by WHATWG) and is extended by the parameters needed for the bootstrap code, AUTH, INVITENUMBER, E-MAIL and NAME.

The joiner (Bob) uses a central element to scan any kind of QR code. When the joiner scans a “setup contact” code, he is prompted if a verified contact should be established to NAME/E-MAIL. If so, the handshake as described in the countermitm-paper is started. The single steps and all parameters are added as “Secure-Join”-headers to the e-mail. Apart from the first, unencrypted mail, these “Secure-Join”-headers are put into the MIME header of the encrypted part.

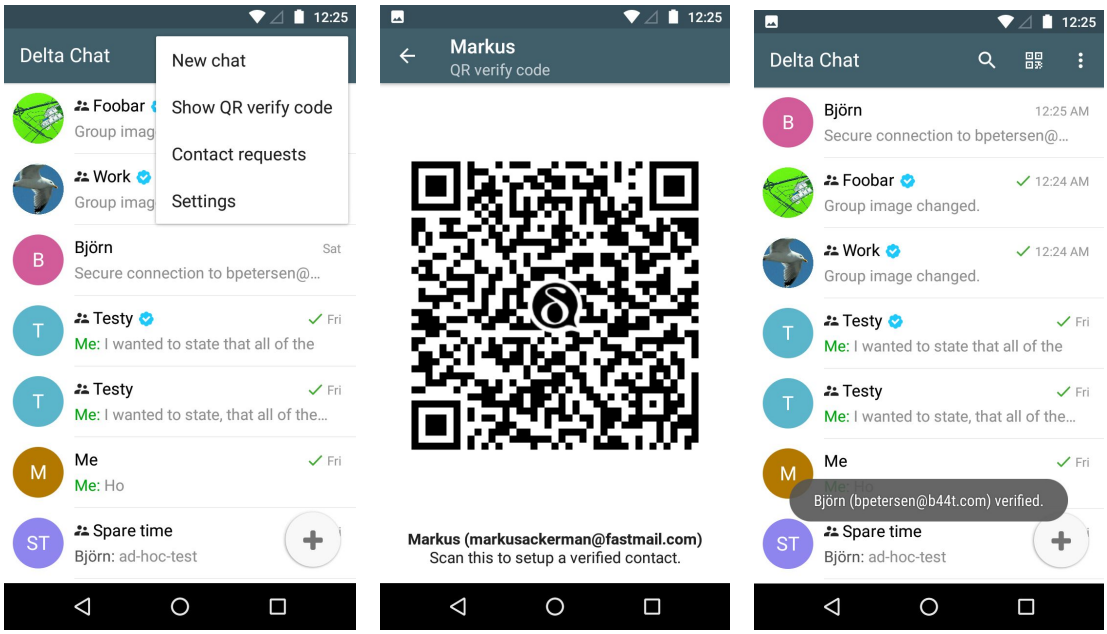
During the handshake, that typically takes several seconds, a progress dialog is shown on the joiner side and the inviter sees the progress as non-disturbing bubbles. This way, it is possible that several people verify a contact with the inviter at the same time.

When the handshake is done, the joiner will be presented the normal, opportunistic chat which shows a system message “Secure connection established”. When, for any reasons, the opportunistic Autocrypt-key does no longer match the verified key, this is also shown as a system message.

As both the inviter and the joiner have a verified key of each other now, this key can be used to create a verified group. These verified groups are similar to normal groups in Delta.Chat with the difference that only verified contacts can be members. To make joining such groups easier, users can *verify+join* in one step shown by a QR code that can be generated from the inviter’s verified group view.

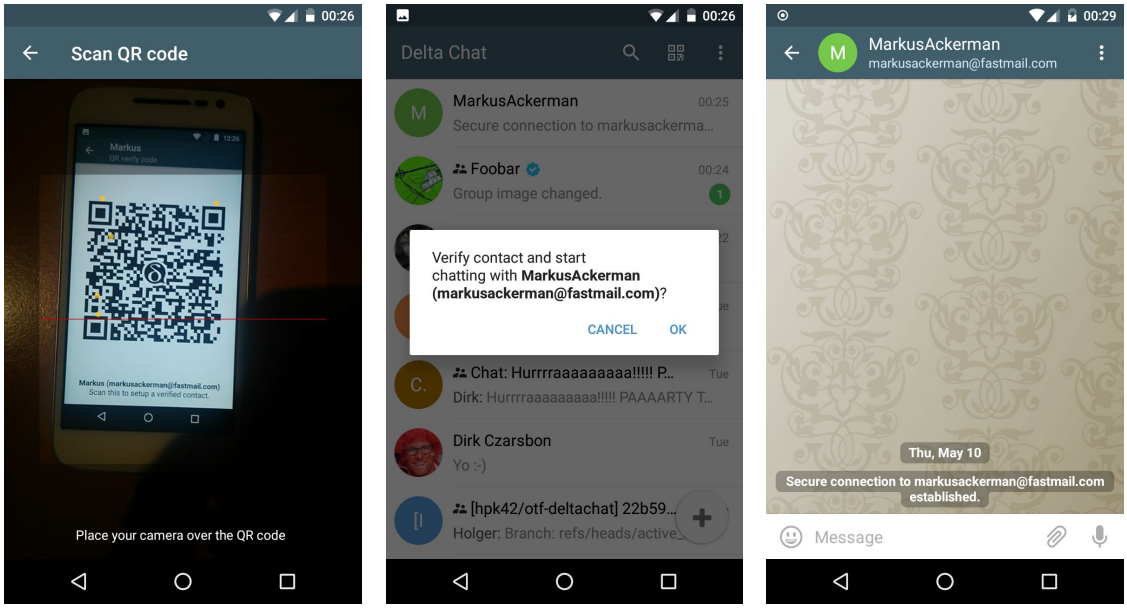
[6] http://countermitm.readthedocs.io/en/latest/new.html#setup-contact-protocol

Setup contact screenshots from the inviter's view



1st screenshot: The inviter clicks on the "Show QR verified" menu entry (upper right).
2nd screenshot: The QR code as displayed on the inviters device; scanned by the joiner (see below)
3rd screenshot: The bubble at the bottom shows that Björn has joined and the chat with Björn is also already visible (first entry, with the system message "Secure connection set up")

Setup contact screenshots from the joiner's view



1st screenshot: The scanning activity (started by the direct icon visible in the title menu) scanning the inviters 2nd screen from above
2nd screenshot: After a successful scan, the joiner is asked if he wants to set up a contact.
3rd screenshot, some seconds later when the handshake is done.

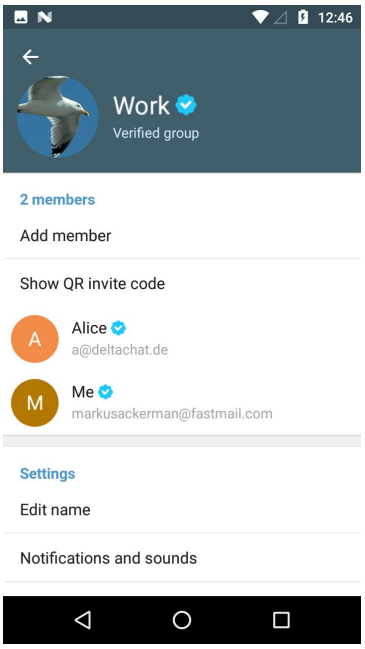
Verified group protocol

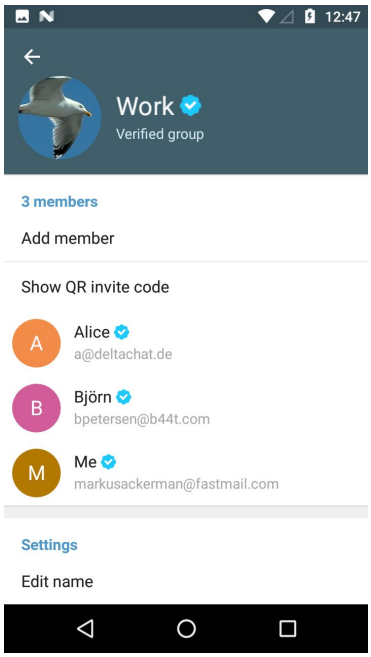
As described above, verified groups are created as normal groups with the difference that only verified contact can be members. To make joining such groups easier, users can verify+join in one step shown by a QR code that can be generated from the inviter's group view.

Such an QR code contains in addition to the "setup contact" QR code the NAME of the group and an internal GROUPID. When the joiner scans such an QR code, Delta.Chat prompts whether to join the group NAME.

In the last step of the handshake protocol, the inviter broadcasts all verified keys of all member through Autocrypt-Gossip. As the inviter is verified by the joiner, these group keys are also treated as verified so that the joiner can send verified messages to all group members from then on.

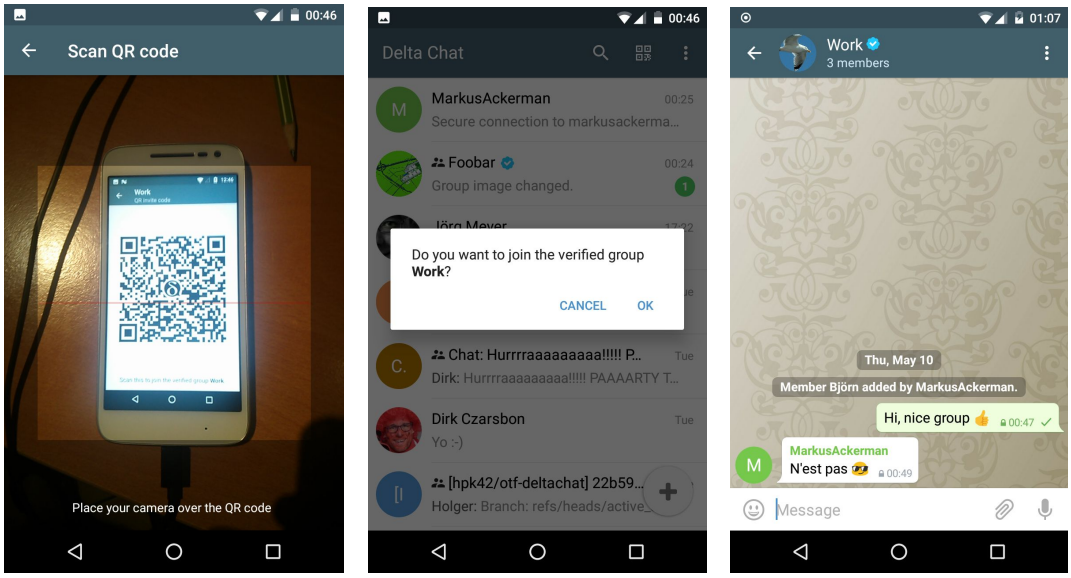
Verified group screenshots from the inviter's view





1st screenshot: A verified group with 2 members
2nd screenshot: The group profile (shown after a tap on the title bar), 2 members and a "Show QR invite code button"
3rd screenshot: Showing the QR invite code for the group
4th screenshot: Björn has joined successfully and is now group member
5th screenshot: The verified group with now 3 members

Verified group screenshots from the joiner's view



1st screenshot: The scanning activity, same activity as for the "setup contact" protocol
2nd screenshot: The inviter's app detected that the QR code belongs to a group invitation and prompts the inviter
3rd screenshot: The joiner has access to the group, including all keys of all prior group members, the group icon and so on

3.2. Releases and summary descriptions

Release versions are given in brackets for each release listed below.

Delta.Chat

Stability, key import and other improvements (0.18.0-0.20.0)

- Check size before sending videos, files and other attachments
- Make sending messages more reliable, fix connection issues
- Speed up by making database-locks unneccessary
- Improve key import
- Detect sending problems related to the message size, show an error and do not try over
- Show message errors in the message info
- Prepare Android bindings update
- Unblock manually blocked members when they are created manually as contact again

Fix verified key implementation after user reported bugs (0.17.3 and 0.17.2)

- Fix system messages appearing twice

- Fix: Use all gossipped verifications in verified groups
- Fix problem with adding formerly uncontacted members to groups
- Unblock manually blocked members when they are created manually as contact again

Fix QR scanning issues (0.17.1)

- Improve QR code scanning screens
- Add a labs-option to disabled the new QR logo overlay

Initial public release with NEXTLEAP protocols (0.17.0)

- implements the “verified contact setup” and “verified group” protocol
- shown “blue checkmarks” beside verified contacts and verified groups
- allows the creation of verified chats through the normal group creation process; the contact list is limited to verified contacts when a verified group is created
- show verified chats with a user in the user’s profiles, so that the user can easily recognize verified
- Move subject and most chat metadata to the encrypted part following the "Memoryhole" proposal

Preliminary support for key verification functionality (0.16.0)

- implements a basic QR scanning function to validate fingerprints from other Delta.Chat’s or any other app following the OPENPGP4FPR:-scheme (eg. OpenKeyChain, APG, GnuPG port)
- if a contact has changed his encryption setups, this is shown as a “system message” in the middle of the opportunistic chats view (*not* in the verified groups, here the key won’t changed)

Conclusion and Future Work

The software releases form the basis of further developments beyond the NEXTLEAP project duration, particularly in the new “Chat over Email” field. Merlinux has secured funding from the OpenTechnologyFund which sustains the work on DeltaChat and NEXTLEAP protocol refinements. Our particular focus in 2019 lies on Eastern Europe and the Ukraine where we develop with methodologies evolved with our partners during the NEXTLEAP project. This methodology works by iteratively and repeatedly designing protocols and implementations together with users, and has been majorly informed and shaped by the work and collaboration with WP3 and WP4. Merlinux also entered several collaborations with other decentralization projects, with E-Mail providers and app developers, and with companies such as OpenXChange which maintains the world’s most popular IMAP-server software and wants to integrate with our released software. All of these collaborations evolved because of our involvement and work with NEXTLEAP and are to result in further real-world deployments of NEXTLEAP protocols.