NTNU | Norwegian University of Science and Technology

PROTOCOL COMPOSITION 1: RSA

TTM4205 – Lecture 15

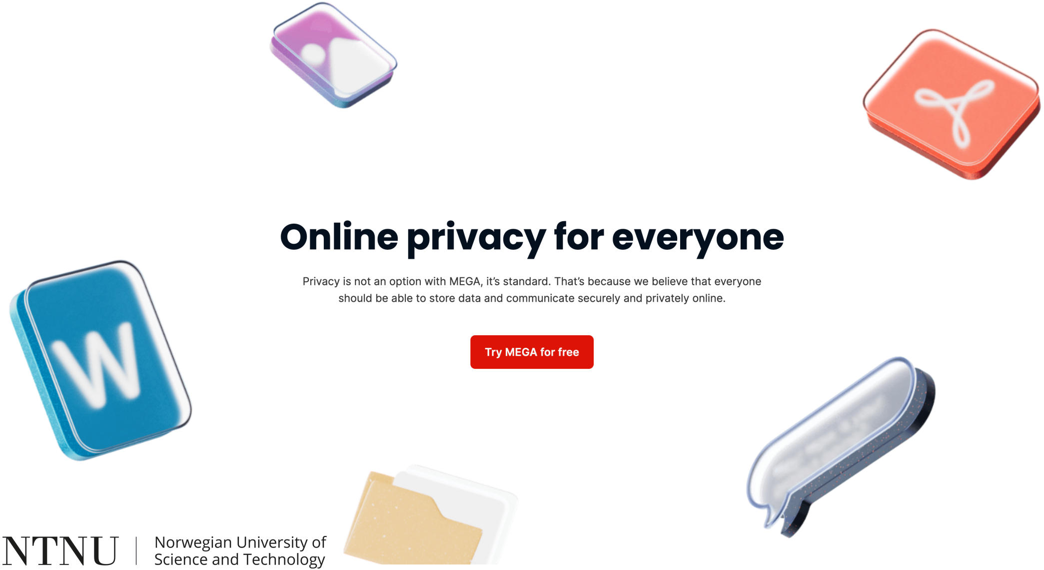
Tjerand Silde

01.11.2024

Contents

Contents

MEGA E2EE Cloud Storage



MEGA E2EE Cloud Storage

- The user encrypt all files locally
- It upload ciphertexts to the cloud
- File-keys are encrypted under master-key
- Master-key is encrypted under password
- The user can log in from anywhere
- The user must sign a challenge on log-in

This is initially a secure infrastructure, but we will see that 1) the choice of ciphers, 2) how they are composed, 3) the lack of integrity checks, 4) custom padding, 5) key-reuse, and 6) server-chosen plaintexts breaks the security.

Contents

MEGA: Malleable Encryption Goes Awry

Matilda Backendal, Miro Haller and Kenneth G. Paterson
Department of Computer Science, ETH Zurich, Zurich, Switzerland
Email: {mbackendal, kenny.paterson}@inf.ethz.ch, miro.haller@alumni.ethz.ch

Figure: https://eprint.iacr.org/2022/959.pdf

MEGA: MALLEABLE ENCRYPTION GOES AWRY

MEGA is a leading cloud storage platform with more than 250 million users and 1000 Petabytes of stored data, which aims to achieve user-controlled end-to-end encryption. We show that MEGA's system does not protect its users against a malicious server and present five distinct attacks, which together allow for a full compromise of the confidentiality of user files. Additionally, the integrity of user data is damaged to the extent that an attacker can insert malicious files of their choice which pass all authenticity checks of the client. We built proof-of-concept versions of all the attacks, showcasing their practicality and exploitability.

Figure: `https://mega-awry.io`

kennyog
@kennyog

MEGA - Malleable Encryption Goes Awry: I'm excited to share details of some new research on the security of @MEGAprivacy. Details at: mega-awry.io (1/28)

11:00 PM · Jun 21, 2022

Figure: `https://twitter.com/kennyog/status/1539352663770509314`

Attacks

Attacks

RSA Key Recovery Attack

MEGA can recover a user's RSA private key by maliciously tampering with 512 login attempts.

Plaintext Recovery

MEGA can decrypt other key material, such as node keys, and use them to decrypt all user communication and files.

Framing Attack

MEGA can insert arbitrary files into the user's file storage which are indistinguishable from genuinely uploaded ones.

Integrity Attack

The impact of this attack is the same as that of the framing attack, trading off less stealthiness for easier pre-requisites.

GaP-Bleichenbacher Attack

MEGA can decrypt RSA ciphertexts using an expensive padding oracle attack.

RSA Key Recovery Attack

RSA Key Recovery Attack

- k_M is an AES key derived from a password

RSA Key Recovery Attack

- k_M is an AES key derived from a password
- $\mathrm{pk}_{\mathrm{share}}$ is a RSA public key $(N = p \cdot q, e)$

RSA Key Recovery Attack

- k_M is an AES key derived from a password
- pk_{share} is a RSA public key $(N = p \cdot q, e)$
- $\text{sk}_{\text{share}}^{\text{encoded}}$ is the corresponding secret key

RSA Key Recovery Attack

- k_M is an AES key derived from a password
- $\mathrm{pk}_{\mathrm{share}}$ is a RSA public key $(N = p \cdot q, e)$
- $\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}$ is the corresponding secret key
- $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ is AES encrypted under k_M

RSA Key Recovery Attack

- k_M is an AES key derived from a password
- $\mathrm{pk}_{\mathrm{share}}$ is a RSA public key $(N = p \cdot q, e)$
- $\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}$ is the corresponding secret key
- $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ is AES encrypted under k_M
- the client is given $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ at every log-in

RSA Key Recovery Attack

- k_M is an AES key derived from a password
- $\mathrm{pk}_{\mathrm{share}}$ is a RSA public key $(N = p \cdot q, e)$
- $\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}$ is the corresponding secret key
- $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ is AES encrypted under k_M
- the client is given $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ at every log-in
- the client is also given $[m]_{\mathrm{pk}_{\mathrm{share}}}$ at log-in

RSA Key Recovery Attack

- k_M is an AES key derived from a password
- $\mathrm{pk}_{\mathrm{share}}$ is a RSA public key $(N = p \cdot q, e)$
- $\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}$ is the corresponding secret key
- $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ is AES encrypted under k_M
- the client is given $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ at every log-in
- the client is also given $[m]_{\mathrm{pk}_{\mathrm{share}}}$ at log-in
- m is a randomly sampled 43 B session ID

RSA Key Recovery Attack

The master secret key is encoded in the following way:

$$\mathsf{sk}_{\mathsf{share}}^{\mathsf{encoded}} \leftarrow l(q)||q||l(p)||p||l(d)||d||l(u)||u||P$$

where $l(\cdot)$ is a length function, q and p are 1024-bit primes, d is the secret RSA exponent, $u = q^{-1} \bmod p$ and P is padding.

RSA Key Recovery Attack

The following happens when the client log in:

RSA Key Recovery Attack

The following happens when the client log in:

- The client derives k_M locally from password

RSA Key Recovery Attack

The following happens when the client log in:

- The client derives k_M locally from password
- The server sends stored $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ to client

RSA Key Recovery Attack

The following happens when the client log in:

- The client derives k_M locally from password
- The server sends stored $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ to client
- The server samples m and sends $[m]_{\mathrm{pk}_{\mathrm{share}}}$ to client

RSA Key Recovery Attack

The following happens when the client log in:

- The client derives k_M locally from password
- The server sends stored $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ to client
- The server samples m and sends $[m]_{\mathrm{pk}_{\mathrm{share}}}$ to client
- The client decrypts $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ then decrypts $[m]_{\mathrm{pk}_{\mathrm{share}}}$

RSA Key Recovery Attack

The following happens when the client log in:

- The client derives k_M locally from password
- The server sends stored $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ to client
- The server samples m and sends $[m]_{\mathrm{pk}_{\mathrm{share}}}$ to client
- The client decrypts $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ then decrypts $[m]_{\mathrm{pk}_{\mathrm{share}}}$
- The client sends m to the server which accepts/rejects

RSA Key Recovery Attack

The following happens when the client log in:

- The client derives k_M locally from password
- The server sends stored $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ to client
- The server samples m and sends $[m]_{\mathrm{pk}_{\mathrm{share}}}$ to client
- The client decrypts $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ then decrypts $[m]_{\mathrm{pk}_{\mathrm{share}}}$
- The client sends m to the server which accepts/rejects
- The server sends all encrypted files to client if accept

$\mathsf{DecSid}([sk_{share}^{encoded}]_{k_M}, [m]_{pk_{share}})$:

Given: encrypted RSA private key $[sk_{share}^{encoded}]_{k_M}$, encrypted message $[m]_{pk_{share}}$

Returns: decrypted and unpadded SID sid'

$sk_{share}^{encoded} \leftarrow \mathsf{AES\text{-}ECB.Dec}(k_M, [sk_{share}^{encoded}]_{k_M})$

$N, e, d, p, q, d_p, d_q, u \leftarrow \mathsf{DecodeRsaKey}(sk_{share}^{encoded})$

$m'_p \leftarrow ([m]_{pk_{share}})^{d_p} \bmod p$

$m'_q \leftarrow ([m]_{pk_{share}})^{d_q} \bmod q$

$t \leftarrow m'_p - m'_q \bmod p$

$h \leftarrow t \cdot u \bmod p$

$m' \leftarrow h \cdot q + m'_q$

$sid' \leftarrow m'[3{:}45]$ // Unpad 43 B SID.

return sid'

Fig. 5. SID decryption during MEGA's client authentication using RSA.

RSA Key Recovery Attack

We can break the system in the following way:

RSA Key Recovery Attack

We can break the system in the following way:

- the secret key $sk_{share}^{encoded}$ is encrypted with AES-ECB

RSA Key Recovery Attack

We can break the system in the following way:

- the secret key $\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}$ is encrypted with AES-ECB
- there is no integrity check for $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$

RSA Key Recovery Attack

We can break the system in the following way:

- the secret key $\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}$ is encrypted with AES-ECB
- there is no integrity check for $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$
- we can edit $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ so that only u changes

RSA Key Recovery Attack

We can break the system in the following way:

- the secret key $\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}$ is encrypted with AES-ECB
- there is no integrity check for $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$
- we can edit $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ so that only u changes
- recover parts of q from chosen m with faulty u

RSA Key Recovery Attack

We can break the system in the following way:

- the secret key $\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}$ is encrypted with AES-ECB
- there is no integrity check for $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$
- we can edit $[\mathrm{sk}_{\mathrm{share}}^{\mathrm{encoded}}]_{k_M}$ so that only u changes
- recover parts of q from chosen m with faulty u
- decrypt all files that the client stored under $[\mathrm{sk}_{\mathrm{pk}_{\mathrm{AES}}}]_{\mathrm{pk}_{\mathrm{share}}}$

RSA Key Recovery Attack

We can recover q from faulty u as follows:

RSA Key Recovery Attack

We can recover q from faulty u as follows:

- If $m < q$ then we get $m'_p = m = m'_q$

RSA Key Recovery Attack

We can recover q from faulty u as follows:

- If $m < q$ then we get $m'_p = m = m'_q$
- Then $t = 0$ and $h = 0$ and $m' = m < 256^{128}$

RSA Key Recovery Attack

We can recover q from faulty u as follows:

- If $m < q$ then we get $m'_p = m = m'_q$
- Then $t = 0$ and $h = 0$ and $m' = m < 256^{128}$
- m' is padded with zeros to 256 bytes

RSA Key Recovery Attack

We can recover q from faulty u as follows:

- If $m < q$ then we get $m'_p = m = m'_q$
- Then $t = 0$ and $h = 0$ and $m' = m < 256^{128}$
- m' is padded with zeros to 256 bytes
- Remove the 211 rightmost bytes

RSA Key Recovery Attack

We can recover q from faulty u as follows:

- If $m < q$ then we get $m'_p = m = m'_q$
- Then $t = 0$ and $h = 0$ and $m' = m < 256^{128}$
- m' is padded with zeros to 256 bytes
- Remove the 211 rightmost bytes
- Then the returned $\mathsf{sid} = \mathrm{m}[3:45] = 0$

RSA Key Recovery Attack

We can recover q from faulty u as follows:

RSA Key Recovery Attack

We can recover q from faulty u as follows:

- If $m \geq q$ then we get $m'_p \neq m \neq m'_q$

RSA Key Recovery Attack

We can recover q from faulty u as follows:

- If $m \geq q$ then we get $m'_p \neq m \neq m'_q$
- Then $t \neq 0$ and $h \neq 0$ since $u \neq q^{-1} \bmod p$

RSA Key Recovery Attack

We can recover q from faulty u as follows:

- If $m \geq q$ then we get $m'_p \neq m \neq m'_q$
- Then $t \neq 0$ and $h \neq 0$ since $u \neq q^{-1} \bmod p$
- Then $m' \neq m > 256^{128}$ with high probability

RSA Key Recovery Attack

We can recover q from faulty u as follows:

- If $m \geq q$ then we get $m'_p \neq m \neq m'_q$
- Then $t \neq 0$ and $h \neq 0$ since $u \neq q^{-1} \bmod p$
- Then $m' \neq m > 256^{128}$ with high probability
- Remove the 211 rightmost bytes

RSA Key Recovery Attack

We can recover q from faulty u as follows:

- If $m \geq q$ then we get $m'_p \neq m \neq m'_q$
- Then $t \neq 0$ and $h \neq 0$ since $u \neq q^{-1} \bmod p$
- Then $m' \neq m > 256^{128}$ with high probability
- Remove the 211 rightmost bytes
- Then the returned $\mathsf{sid} = \mathsf{m}[3:45] \neq 0$

RSA Key Recovery Attack

This means that we learn 1 bit of information each time, and can use a binary search between $[2^{1023}, 2^{1024})$ to find q in at most 1023 queries i.e. learn one bit each time the client tries to log in.

Using an improved lattice-attack similar to the attack on ECDSA allowed for a reduction to 512 queries total, drastically reducing the number of times a client needs to log in to be vulnerable.

Other Attacks

The re-use of keys also allowed for decryption oracles, the custom RSA padding P allowed for Bleichenbacher attacks, lack of integrity checks allowed for uploading malicious material, and more.

They added HMAC checks, updated padding and updated the key-hierarchy after this work, but claimed that 512 log-in attempts was too much for this to be a realistic attack…

Contents

The Hidden Number Problem with Small Unknown Multipliers: Cryptanalyzing MEGA in Six Queries and Other Applications

Keegan Ryan and Nadia Heninger

University of California, San Diego
kryan@eng.ucsd.edu,nadiah@cs.ucsd.edu

Figure: https://eprint.iacr.org/2022/914.pdf

We actually learn more than 1 bit per query. The sid is of size 43 bytes, and it leaks much more information across the different queries. With a lot of pre-processing, it was shown that we can recover q in only 6 queries (!). The lattice-attack achieving this is out of scope for this lecture.

Contents

Caveat Implementor! Key Recovery Attacks on MEGA

Martin R. Albrecht[1], Miro Haller[2], Lenka Mareková[3], and Kenneth G. Paterson[2]

[1] King's College London
martin.albrecht@kcl.ac.uk
[2] Applied Cryptography Group, ETH Zurich
kenny.paterson@inf.ethz.ch, miro.haller@ethz.ch
[3] Information Security Group, Royal Holloway, University of London
lenka.marekova.2018@rhul.ac.uk

Figure: https://eprint.iacr.org/2023/329.pdf

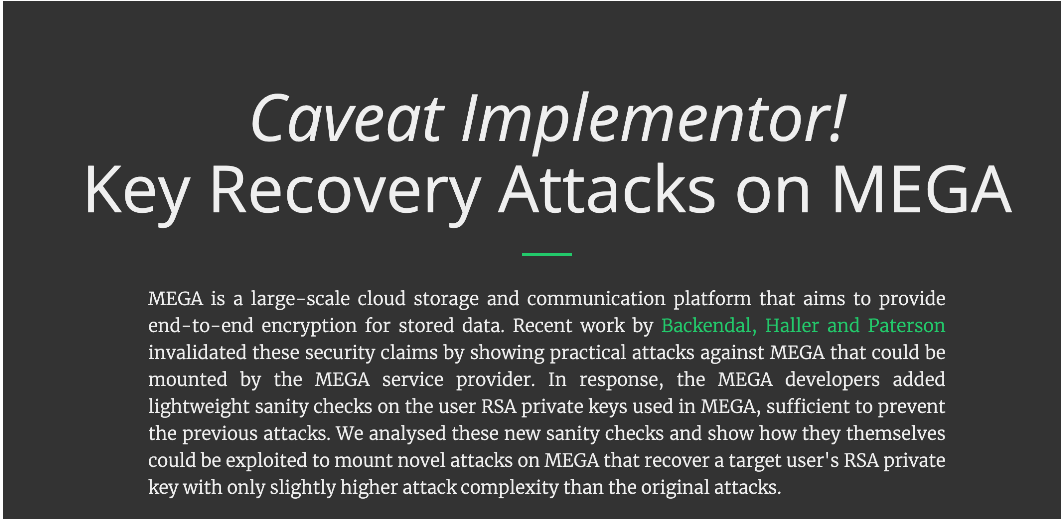


Figure: `https://mega-caveat.github.io`

kennyog
@kennyog

In a new paper, @martinralbrecht, @M__Haller, Lenka Mareková, and I took a fresh look at @MEGAprivacy. TL;DR: we broke the fixed version with attacks that can recover user RSA private keys and file keys. Paper and more at: mega-caveat.github.io (1/16)

Figure: `https://twitter.com/kennyog/status/1632718211476078592`

New Attacks

They came up with the following new attacks:

New Attacks

They came up with the following new attacks:

- Reducing the original attack from 512 to 2 queries

New Attacks

They came up with the following new attacks:

- Reducing the original attack from 512 to 2 queries
- Exploiting re-encryption with adversarial keys

New Attacks

They came up with the following new attacks:

- Reducing the original attack from 512 to 2 queries
- Exploiting re-encryption with adversarial keys
- Error messages that reveal more information

New Attacks

They came up with the following new attacks:

- Reducing the original attack from 512 to 2 queries
- Exploiting re-encryption with adversarial keys
- Error messages that reveal more information
- Still using AES-ECB because it is "cheaper"

New Attacks

They came up with the following new attacks:

- Reducing the original attack from 512 to 2 queries
- Exploiting re-encryption with adversarial keys
- Error messages that reveal more information
- Still using AES-ECB because it is "cheaper"
- Key-overwriting attacks from lacking integrity

Contents

Four Attacks and a Proof for Telegram*

Martin R. Albrecht[1], Lenka Mareková[2], Kenneth G. Paterson[3], and Igors Stepanovs[3]

[1] King's College London
martin.albrecht@kcl.ac.uk
[2] Information Security Group, Royal Holloway, University of London
lenka.marekova.2018@rhul.ac.uk
[3] Applied Cryptography Group, ETH Zurich
{kenny.paterson,istepanovs}@inf.ethz.ch

31 March 2023

Figure: https://eprint.iacr.org/2023/469.pdf

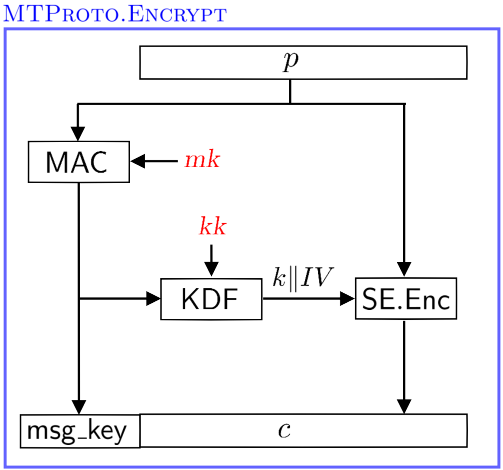
MTProto

The **MTProto** protocol is not well-studied:

- **2013: Telegram launched** with MTProto 1.0.
- **2016: Jakobsen** and **Orlandi** showed that MTProto 1.0 is not CCA-secure.
- **2017: Telegram released MTProto 2.0** that addressed the security concerns.
- **2017: Sušánka** and **Kokeš** reported an attack based on improper validation in the Android client.
- **2018: Kobeissi** reported input validation bugs in Telegram's Windows Phone client.
- **2020: Miculan** and **Vitacolonna** proved MTProto 2.0 secure in a symbolic model, assuming ideal building blocks.

Figure: `https://iacr.org/submit/files/slides/2022/rwc/rwc2022/60/slides.pdf`

MTProtoEncrypt



MTProtoEncrypt

supplied by attacker

If (msg_length > length) then . . . // **Android**

Outcome of comparison depends on 32 bits on msg_length.

If comparison fails: two conditional jumps added.

If (msg_length > 2^{24}) then . . . // **Desktop**

Outcome of comparison depends on 8 bits on msg_length.

If comparison fails: MAC verification is omitted.

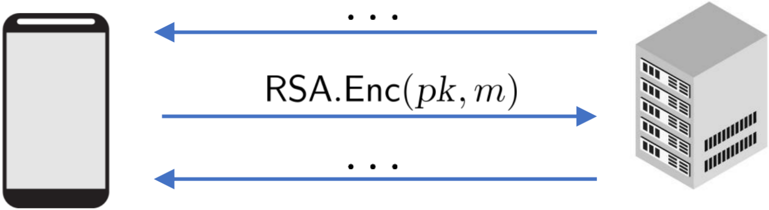
If not ($12 \le \ell -$ msg_length ≤ 1024) then . . . // **iOS**

Outcome of comparison depends on 32 bits on msg_length.

If comparison fails: MAC verification takes a shorter input.

MTProtoEncrypt

We attack **Telegram**'s key exchange.



Telegram uses textbook **RSA** encryption.

$$m := \mathsf{SHA\text{-}1}(\mathsf{data}) \| \mathsf{data} \| \mathsf{padding}$$

Four Attacks

- Message reordering (lack of metadata authentication)
- Re-encryption of dropped messages lead to CPA attacks
- Timing attack against encrypt and mac using AES-IGE
- RSA padding oracle using textbook RSA with SHA-1

Future Work

Large parts of **Telegram**'s design remain unstudied:

Secret chats (including encrypted voice and video calls).
The key exchange.

Multi-user security.
Forward secrecy.

Telegram Passport.
Bot APIs.

The higher-level message processing.
Control messages.

Encrypted CDNs.
Cloud storage.

These are pressing topics for future work.

Questions?