Asyncronous Mobile P2P Relay

December 2015
- Max Skibinsky (max@skibinsky.com)
- Yevgeniy Dodis (dodis@cs.nyu.edu)

---

1. Introduction

To create a practical mobile peer-to-peer network we have to solve a number of challenges that are distinctly different from the usual patterns of desktop-based p2p networks.

To minimize smartphone battery use, a persistent mobile application will be kept by its mobile OS in a suspended state most of its lifetime. A peer-to-peer application that is waiting for external communications might be unloaded by the user or suspended by the host OS at any moment. To make the p2p network reliable we will need a backbone of stable nodes that can relay communications between endpoint devices, similar to Skype's dedicated super-nodes[1]. Once an "always-on" relay backbone is present, mobile p2p application can establish a reliable communication channel between end-point devices. The relays will receive encrypted communications from an end-point mobile sender and cache these communications until the recipient device is woken up (by internal events or a notification from the host OS) to collect them. Such a relay network should provide a secure asynchronous transport layer to any p2p mobile app regardless of app specifics.

Let's consider how end-points should communicate with these relays. With an extremely high number of mobile devices[2], it is implausible to create a schema that depends on the reliable device pre-registration for each relay. A few million devices joining the network with little notice could be a legitimate use case (fast growing and popular new mobile app), or it might be a DDOS attack. A popular application can have an install base reaching into hundreds of millions of users, which would make it prohibitively complex to add and provision new relays into the network. To make relay nodes lightweight and dynamic, we should allow any device to communicate with any relay with no pre-conditions, yet introduce a non-zero cost of a potential DOS attack - relays should thus require a "proof of work" handshake (made increasingly difficult as the relay's load factor increases) from each device to establish a new communication session.

Since any relay can communicate with any device, that makes relays fully interchangeable. In turn, we obviously would want for the relay backbone to become more robust with as additional relays are added. At the same time, growth of the backbone network will increase the exposure of end-point device communications to an increased number of parties. Therefore, each relay should provide strong privacy: it should be theoretically impossible for any relay to recover the contents of p2p communications passing through it, and relays should never store communication contents in any form of permanent storage.

Finally, the relay should be reasonably robust in scenarios of partially broken or misconfigured installations. In the broader community, relay instances might be deployed by hobbyists without proper security training. When possible, we should add reasonable safeguards protecting end-point information passing through the relay even if the given relay is mis-configured and leaks some information about its operations (such as log files, misconfigured TLS, etc).

1.1 Relay Node Requirements

Lets summarize all the requirements of an asynchronous relay node:

- **Universal**: All the relay nodes should be mutually interchangeable and operate in a global address space. Any mobile device should be able to contact any node to pass private communications to any other mobile device without a pre-existing setup or registration with that relay.
- **Encrypted end-to-end**: It should be cryptographically impossible for the relay node to decrypt the traffic passing though it between endpoint devices - even if the relay is taken over by an external agency.
- **Ephemeral**: Relay nodes should not store anything in permanent storage, and should operate only with *memory-based* ephemeral storage. The relay should act as an encrypted memory cache shared between mobile devices. Key information should be kept in memory only and erased within minutes after completing the session. Encrypted communications should be kept in memory for future collection and should be erased after a few days if not collected by the target device.[3]
- **Well-known relays**: A deployed relay's URL/identity should be well known and proven by a TLS certificate. Applications might implement certificate pinning for well-known relays of their choosing. A mobile app could keep a list of geographically dispersed relays and use a deterministic subset of them to send & receive asynchronous communications to/from another mobile device.
- **Identification privacy**: After establishing temporary keys for each session, a relay node should never store long-term identity keys of the endpoint devices. When the protocol requires the verification of public key ownership, these operations should happen in memory only and should be immediately erased afterward.[4]
- **Resilient**: Relays should be reasonably resilient to external takeover and network traffic intercept. Such a takeover, if successful, should only expose communication metadata, but not the content of any communications. Minor mis-configurations of a relay node, (such as leaking log files, etc.), during deployment by more casual users should not lead to a catastrophic breakdown of communication privacy.
- **Private nodes**: Power users should have the option to deploy their own personal

relay nodes and have the ability to add them into the configuration of mobile applications that are reliant on this kind of p2p network.

2. Relay Protocol

2.1 Overview

Each mobile device (Alice, Bob, etc.) has a pair of *long term identity communication keys*: secret key **comm_sk** and public key **comm_pk**. When a p2p application establishes a communication circle of specific devices it wants to talk with, it will exchange device identity public keys between each other via the usual mobile communications methods (text, email, chat). Communications with a relay (Rebecca) happen after an identity key exchange has already taken place between Alice and Bob. Endpoint applications can use these communication keys either for direct end-to-end messaging if perfect forward secrecy is not a factor, or as starting keys in forward-secure ratchet schema[5]. Communications between mobile device and relay instance will always be forward secure: after a session is established both sides will use temporary keys created for that specific session, and both sides delete the temporary keys after the session.

All encryption and authentication operations are based on the NaCl library [6] PKE functions. All hashes are performed as $\mathbf{h_2(m) = sha256(0_{32},sha256(m))}$[7]. A relay identifies devices by the h_2 hash of the device's public *comm_pk* key. The hash of the device's *comm_pk* is the permanent identification of that device for its peers. We refer to the $h_2(comm\_pk)$ hash of the public key as **hpk** for a given device.

Hashing of the public long term communication keys forms a global address space between all relay nodes. A mobile device can leave communications for a specific *hpk* on any relay, and in turn, check for its own communications for its device *hpk* on any relay. Relays have no methods of verifying "valid" *hpk* destinations, and will hold communications for any *hpk* given by the originator. If communications are not collected, they are flushed from memory within 3 days. If a relay is overloaded, its up to the relay administrator to start deleting stale communications faster (maintaining the memory-only storage policy) or to start caching new communications to hard drive (maintaining reliability during high load).[8]

As a measure against simple denial of service attacks, relay communications are a two step process. In the first step, any client initiates a session handshake by sending a random string to the relay and getting another random string in response. This random pair will identify the session until both sides exchange session keys. Each relay has a **difficulty** setting, that can be set by relay administrator or rise dynamically if the relay is under heavy load. The default *difficulty* is 0 and in such cases the "proof of work" step is skipped. In these cases, the client verifies initiated session by responding with a hash of the client and relay strings $h_2(client\_token,relay\_token)$ - a simple confirmation that it received the relay random string. Such verification doesn't require access to the main relay systems (key generation & communication storage) and can be done at the load balancer level, or at a separate, isolated tier of public facing instances, which can be brought online quickly during periods of increased load.

When the relay *difficulty* is *above zero*, in addition to a random string, the relay will send its difficulty value [1..255] after the handshake random string. Now, instead of sending the h_2 hash, client will have to send a unique 32 bytes *difficulty nonce* instead: a value such that *h_2(client_token,relay_token,diff_nonce)* results in *difficulty* count of *leading zero bits* in the hash result. For example, a difficulty setting of 8 requires the first byte of the hash result to be zero. A simple way to think about this algorithm: at the default *difficulty* of 0 any nonce value would be correct, so the client just skips the proof of work loop and responds with the "minimal" work unit of calculating *h_2(client_token,relay_token)*. If proof of work is enabled by relay admin, client does need to calculate proper *diff_nonce* for *h_2(client_token,relay_token,diff_nonce)* to satisify leading zeroes condition and send it back to complete the handshake.

In the second part of the handshake, after the relay verifies the token exchange, it will generate ephemeral keys for the session. The client will generate its own session keys and, after the key exchange, prove ownership of the long term identity key corresponding to **hpk** (all these communications are encrypted with the session keys). Once this ownership has been proven, the relay will associate in a limited time window the current client session key with the (now provenly owned) *hpk* and execute client commands to check, upload, and delete communications for *hpk* that are sent using that client session key.

In general, the relay should be well configured, with properly setup TLS that accepts only https requests and a logging policy that doesn't save the content/metadata of POST requests.

2.2 Protocol Schema

Mobile device Alice **A** communicates with relay Rebecca **R**.

2.2.1 Schema Defaults

- Each session token or public key is shared once in plaintext in the **A** □ **R** channel, and used in hashed form in all the following communications.
- Each encrypt/decrypt operation includes a random 24 byte NaCl nonce with the first 8 bytes being the big endian epoch time in seconds as protection from replay attacks.

2.2.2 Schema Legend

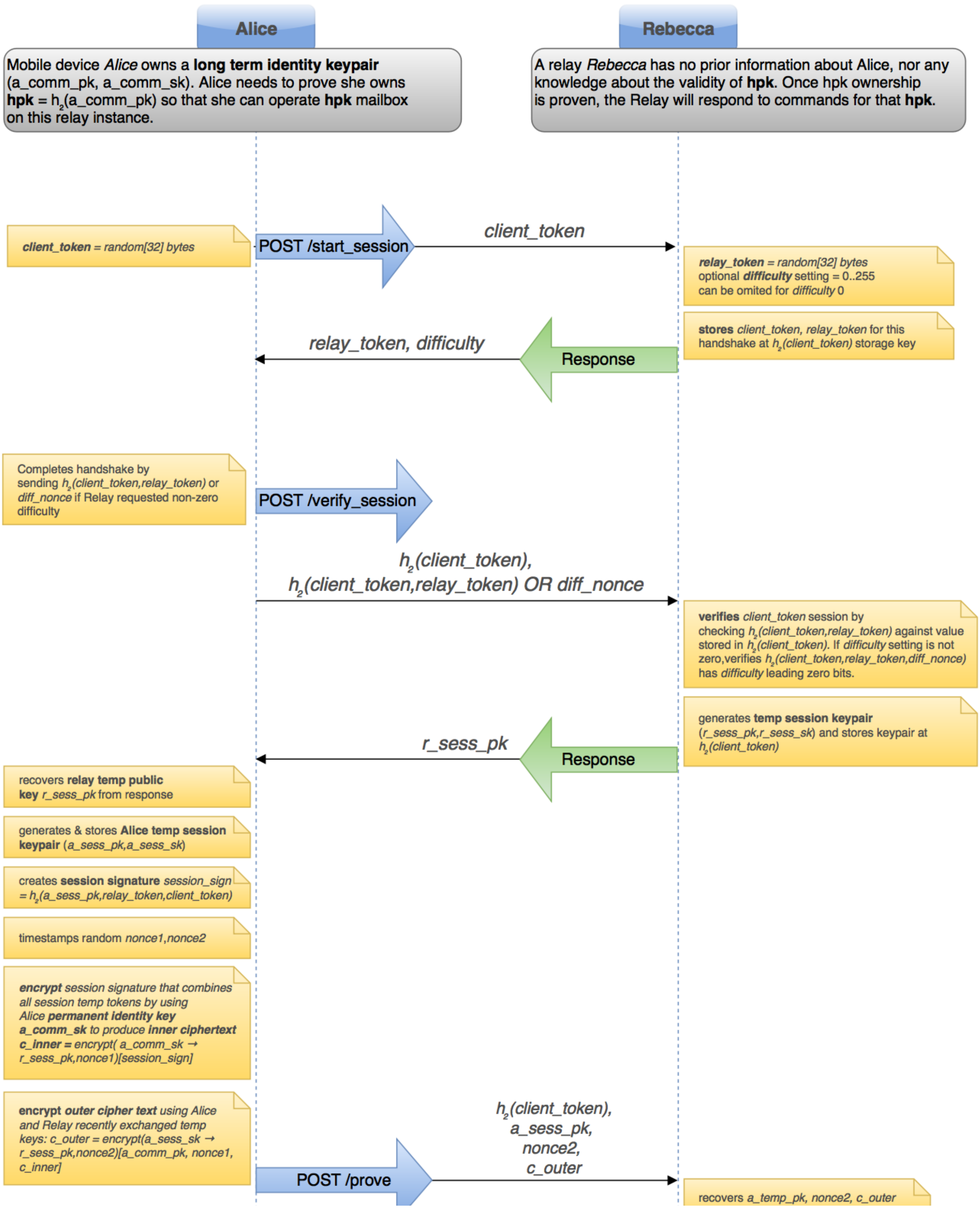
- **A → R** : Alice **A** sends to Rebecca **R**
- **A ← R**: Rebecca **R** sends to Alice **A**

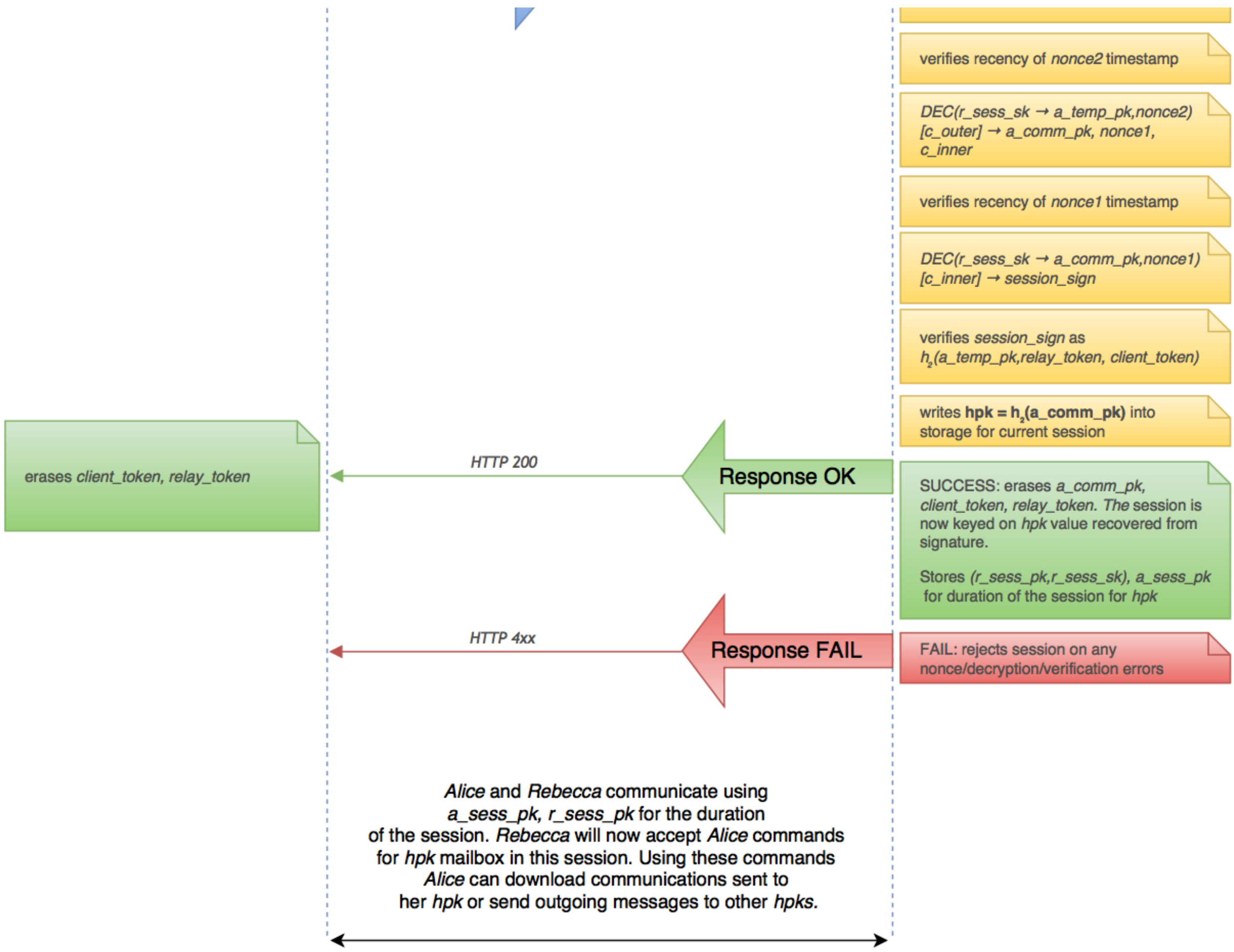
Alice **A** owns:

- (a_comm_pk, a_comm_sk) - a long term identity communication key pair
- hpk = h_2(a_comm_pk)

2.3 Communication Protocol

2.3.1 A starts new session with R:



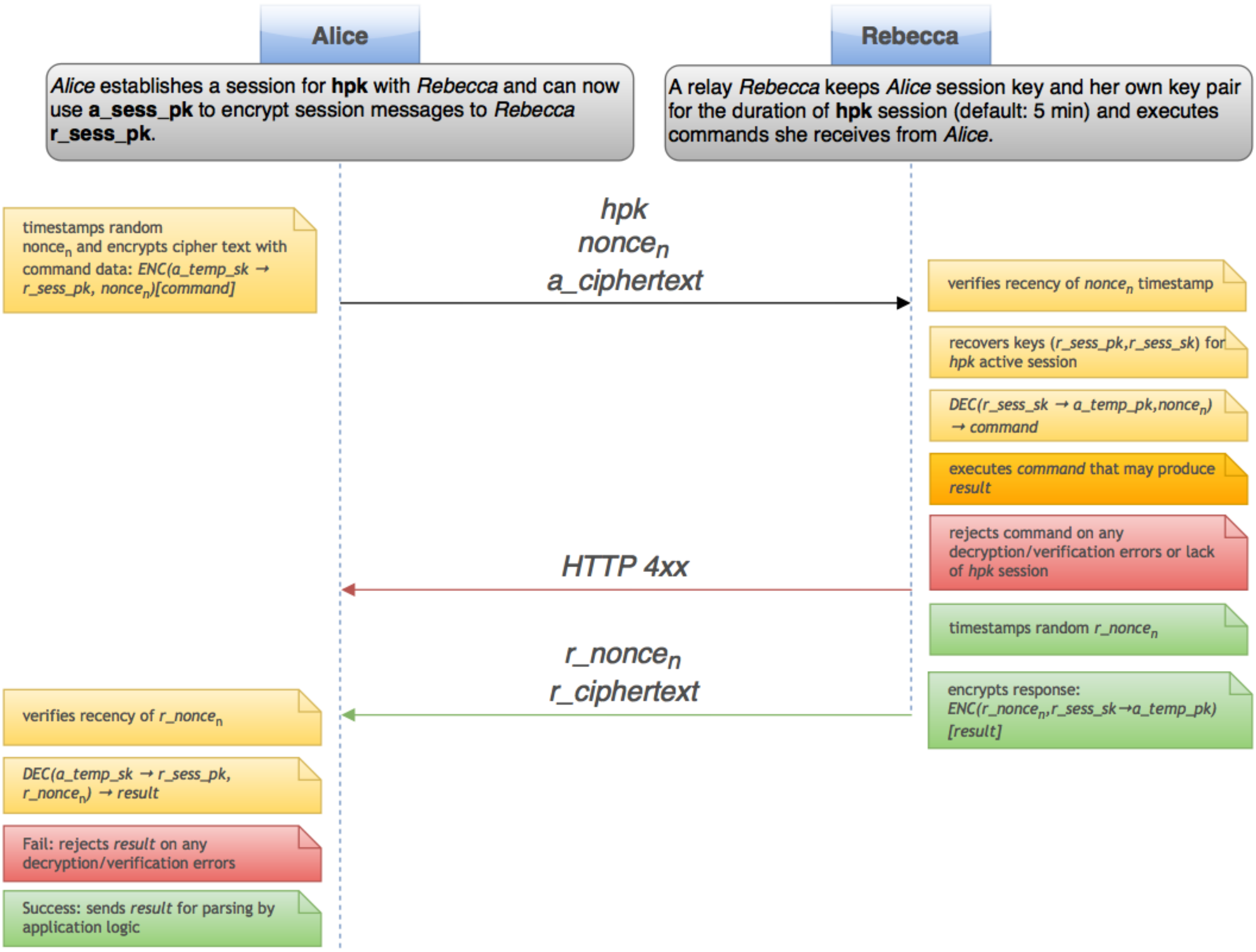


1. **A** generates *client_token* = random 32 bytes
2. **A** sends *client_token* → **R**
    - **R** stores *client_token* for handshake session
    - **R** generates *relay_token* = random 32 bytes
    - **A** ← *relay_token*, *difficulty* responds **R**
3. **A** sends *h_2(client_token), h_2(client_token,relay_token) OR diff_nonce* → **R**
    - **R** verifies *h_2(client_token,relay_token)* from *relay_token* on *client_token* session for zero difficulty. If *difficulty* is not zero it checks that first *difficulty* bits of *h_2(client_token,relay_token,diff_nonce)* are zero.
    - **R** generates and stores temporary session keys (*r_sess_pk,r_sess_sk*)
    - **A** ← *r_sess_pk* responds **R**
4. **A** gets *r_sess_pk* from **R** response
5. **A** generates & stores own temporary session keys (*a_sess_pk,a_sess_sk*)
6. **A** creates *session_sign = h_2(a_sess_pk,relay_token,client_token)*
7. **A** timestamps random *nonce1*,*nonce2*
8. **A** *c_inner = ENC(a_comm_sk → r_sess_pk,nonce1)[session_sign]*
9. **A** *c_outer= ENC(a_sess_sk → r_sess_pk,nonce2)[a_comm_pk, nonce1, c_inner]*
10. **A** sends *h_2(client_token), a_sess_pk , nonce2, c_outer* → **R**
    - **R** recovers *a_sess_pk, nonce2, c_outer*
    - **R** verifies recency of *nonce2* timestamp
    - **R** *DEC(r_sess_sk → a_sess_pk,nonce2)[c_outer] → a_comm_pk, nonce1, c_inner*
    - **R** verifies recency of *nonce1* timestamp
    - **R** *DEC(r_sess_sk → a_comm_pk,nonce1)[c_inner] → session_sign*

- **R** verifies *session_sign* as h_2*(a_sess_pk,relay_token, client_token)*
- **R** writes **hpk == h_2(a_comm_pk)** for this session
- **R** rejects session on any decryption/verification errors
- **R** on successful session:
  - **R** erases *a_comm_pk, client_token, relay_token*
  - **R** stores *(r_sess_pk,r_sess_sk), a_sess_pk, hpk* for duration of the session for *hpk*

11. **A** on successful session:
    - **A** erases *client_token, relay_token*
    - **A** stores *(a_sess_pk,a_sess_sk), r_sess_pk* for duration of the session with **R**

2.3.2 A sends command (a command-specific data-block) number n to R



1. **A** timestamps random nonce_n
2. **A** *hpk, nonce_n , ENC(a_sess_sk → r_sess_pk, nonce_n)[command]* → **R**

- **R** verifies recency of *nonce_n* timestamp
- **R** recovers keys (*r_sess_pk,r_sess_sk*) for *hpk* active session
- **R** *DEC(r_sess_sk → a_sess_pk,nonce$_n$) → command*
- **R** executes *command* that may produce *result* (a command-specific response data-block)

- **R** timestamps random *r_nonce_n*
- **A** ← *r_nonce_n, ENC(r_nonce_n ,r_sess_sk →a_sess_pk)[result]* **R**
- **R** rejects command if any decryption/verification errors or lack of *hpk* session.

3.**A** verifies recency of *rnonce_n*
4. **A** *DEC(a_sess_sk → r_sess_pk, rnonce_n) → result*
5. **A** rejects *result* if any decryption/verification errors. On success **A** sends *result* to higher level application logic for processing.

3. Relay HTTP Specification

3.1 Alice initiates session

POST /start_session/
Authentication: none. Any device can make this request.

Alice generates a random 32 byte *client_token* and makes a *POST /start_session/* request with *base64(client_token)* string as the single line body.

- line 1: **base64(client_token)**

Response

- Rebecca generates a 32 byte **relay_token**
- Rebecca stores the following values in the memory caching layer at h_2*(client_token)* key with a 1 min expiration (config param):
  - client_token
  - relay_token
- Rebecca responds with a base64 first line of *relay_token* as response. Plain text *difficulty* value is the second line of this response.
- line 1: **base64(relay_token)**
- line 2: **difficulty**

Rebecca is IP-throttled to 1 handshake request per 10 sec (config param) for a given IP.

Notes
- *difficulty* can be in [1..255] range or zero
- The same *client_token* requests should return the same *relay_token* until the handshake attempt expires (1m).
- A different *client_token* generates a different *relay_token*

POST /verify_session/

Alice completes the handshake via POST to /verify_session/

- line 1: **base64(h_2(client_token))** of the same *client_token* used in */start_session* request
- line 2: **base64(h_2(client_token,relay_token))** OR **base64(diff_nonce)**

Rebecca checks for a valid handshake by reading from the *h_2(client_token)* memory only cached entry and calculating *h_2(client_token,relay_token)* for a zero *difficulty* setting. For any non-zero difficulty, Rebecca verifies that the first *difficulty* bits of *h_2(client_token,relay_token,diff_nonce)* are 0.

If the handshake is correct, Rebecca generates a session key pair **(r_sess_pk,r_sess_sk)**. Rebecca caches this key pair in memory at *h_2(client_token)*. This key pair should expire in 5 min (config param).

Response:
- Rebecca responds with a plaintext base64 encoded *r_sess_pk* as a single line response.
- line 1: *base64(r_sess_pk)*

Session state
Cached in memory for 5 minutes (config param) at the *h_2(client_token)* key:

- relay key pair: *(r_sess_pk,r_sess_sk)*
- *client_token*
- *relay_token*

Notes
- The same *client_token* requests should return the same session key-pair until the handshake expires (5 min).
- A different *client_token* verification returns a different key pair.
- If there is no cached entry for *h_2(client_token)*, the request immediately fails. Rebecca can use verification of fixed size first line of POST body as a quick rejection filter for malformed requests.

Deployment
- initial handshake requests with zero difficulty can be verified at the load balancer level.
- instances that verify handshake requests can be isolated from the rest of the infrastructure.
- these instances require relatively little resources since key generation doesn’t happen until after the handshake.
- Verification instances can be brought online if the relay is under heavy load and protect instances serving verified connections.

End State

After this exchange Rebecca has shared a temporary session key with Alice.

3.2 Alice proves ownership of hpk

POST /prove_hpk

Authentication:

- previous successful handshake for *client_token*
- ownership of private identity key controlling *hpk*.

Alice now has *r_sess_pk* and *relay_token* and can prove ownership of *hpk* while the handshake for *client_token* is active.

- Alice generates a temporary key pair **(a_sess_pk,a_sess_sk)**
- Alice writes **h2(client_token)** as *1st line* (base64) of the POST
- Alice writes **a_sess_pk** as *2nd line* of POST (base64)
- Alice picks 24 byte **nonce_outer**. First 8 bytes are the UTC timestamp and rest random.
- Alice writes **base64(nonce_outer)** as *3rd line* of POST
- Alice creates **nonce_inner** in the same way.
- Alice creates 32 byte session signature as **h_2(a_sess_pk,relay_token,client_token)**
- Alice encrypts the signature with **crypto_box(nonce2, r_sess_pk, a_comm_sk)** resulting in **cyphertext_inner**
- Alice creates a JSON object[9]:

```
JSON = {
 nonce: **nonce_inner**,
 pub_key: **a_comm_pk**,
 sign: **cyphertext_inner**
}
```

Alice encrypts this JSON object with **crypto_box(nonce, r_sess_pk, a_sess_sk)** resulting in **cyphertext** and writes **base64(cyphertext)** as *4th line* of POST

Alice POST request by line numbers:

- line1: *base64(h_2(client_token))*: same *client_token* used to receive *r_sess_pk*, serves as the handhshake id.
- line2: *base64(a_sess_pk)*: Alice temporary session key using previously exchanged *relay_token*
- line3: *base64(nonce_outer)*: Timestamped nonce.
- line4: *base64(crypto_box(JSON, nonce_inner, r_sess_pk, a_sess_sk))*: Outer crypto-text.

Rebecca receives a double encrypted envelope. The outer envelope is encrypted with

the session key and the internal payload is encrypted with Alice's long term identity *communication key* against Rebecca's *session key*.
Rebecca has no information as to whether *hpk* represents a real hash. Rebecca only checks if there are any communications for *hpk* in its current in-memory records.

- Rebecca recovers h_2*(client_token)* from the first line, and retrieves the session state from memory caching service. If there is no session state the request fails.
- Using *relay_token* from the session state, Relay de-masks *a_sess_pk*
- Rebecca checks the timestamp section of the outer nonce. Rebecca will reject packets with a time difference higher than 3 min (config param). To account for any potential mobile phone clock desynchronization, we keep the timestamp validity window relatively large.
- Rebecca caches all *nonces* received in memory for 10 min (config param). Relay rejects any communication where *nonce* is reused to prevent replay attacks within timestamp validity window.
- Rebecca decrypts outer cipher-text **crypto_box_open(nonce_outer, a_sess_pk, r_sess_sk**) and recovers the JSON object with *a_comm_pk*.
- Rebecca checks the timestamp section of inner nonce. Rebecca will reject packets with a time difference higher than 3 min (config param).
- Rebecca can decrypt the inner cipher-text with **crypto_box_open(JSON.nonce, JSON.pub_key, r_sess_sk)** and verify the session signature h_2*(a_sess_pk,relay_token,client_token)*.
- If the communication passes all of the above confirmations:
  - Rebecca writes *hpk* = h_2*(a_comm_pk)* into current session memory storage.
  - Rebecca writes the total number of communications for *hpk* in response body - or zero if there is no cached entry for *hpk*. Alice can use that information to determine if she needs further communication with Rebecca.
  - Rebecca moves all cached data from h_2*(client_token)* key entry to *hpk* entry and sets expiration to 10 min (config param). After the transfer h_2*(client_token)* the cached contents are deleted.

Success Response

- *line 1*: plaintext count of communications for *hpk*.

The decryption protocol verifies that:

- Alice is in possession of (*a_comm_pk,a_comm_sk*) keypair
- *a_comm_pk* hashes into *hpk* which becomes the new session id for Rebecca.
- Alice's request is recent due to the *nonce* timestamp checks
- Alice's request is for this relay since it verifies recently given *relay_token* mask
- Alice's *a_comm_pk* is transferred over the wire in encrypted form over (r_sess_pk, a_sess_sk) key pair.
- The session signature is unique for the combination of (a_sess_pk,relay_token,client_token)

After Rebecca verifies all the checkpoints above, Alice should be allowed to downloaded traffic for *hpk* since she has proven ownership of *a_comm_sk* corresponding to the *a_comm_pk* which hashes to *hpk*

Rebecca can store *a_sess_pk* as the authenticated key for *hpk* and will accept other requests referring to hpk for the duration of the (*r_sess_pk,r_sess_sk*) key pair. Rebecca will decipher traffic for *hpk* using (*a_sess_pk,r_sess_sk*) until *r_sess_sk* expires in 10 minutes.

Session state
When Rebecca has established a session for *hpk* she keeps the following in a memory only cache associated with *hpk*

- Rebecca's session key pair (*r_sess_pk,r_sess_sk*)
- *a_sess_pk*
- **deletes** *relay_token, client_token, a_comm_pk* from memory
- Alice keeps the session keys *(a_sess_pk,a_sess_sk),r_sess_pk* for the same duration.

Notes
Rebecca will respond to a new *new_session, verify_session* to establish another session, which in turn can be used to verify *hpk* ownership, even if an *hpk* session is already established. If *hpk* is proven on another session, which implies another temporary key pairs on both sides, the new session state with new key information overwrites all cached state associated with the existing *hpk* session.

End State
Rebecca decrypts and encrypts traffic for the device with established *hpk* session for the *(a_sess_pk,r_sess_sk)* key pair.

3.3 Alice sends relay commands

POST /command
Authentication: Owner of private key for *hpk*.

- Alice communicates by encrypting her traffic to/from relay with (**r_sess_pk,a_sess_sk**)
- Rebecca communicates by encrypting responses to/from Alice with (**a_sess_pk,r_sess_sk**)
- *Nonces* are always timestamped and checked by both sides to match current UTC time within 3 min (config param). Rebecca caches in memory all *nonces* received for 10 min (config param). Rebecca rejects any communication where *nonce* is reused to prevent replay attacks within the timestamp validity window.

Alice creates a JSON data structure that must have the command name field **cmd**. Alice encrypts this JSON data structure with (**r_sess_pk,a_sess_sk**) to get *cipher-text*

Alice's POST to relay by lines:

- line1: base64(*hpk*)
- line2: base64(*timestamped nonce*)
- line3: base64(*cipher-text*)

Relay will:

- Checks that there is an active session state for *hpk*
- Check the *nonce* timestamp
- time-check key expiration
- decrypts the second line with *(nonce, a_sess_pk, r_sess_sk)* with the key taken from the established session for *hpk*
- recovers the JSON data structure with **cmd**

3.3.1. 'count' command

Rebecca expects a JSON object with only '*cmd*' field with '*count*' value. Rebecca responds by encrypting the following JSON data structure

```
 { count: 0 #number of communications for hpk }
```

Rebecca's response:

- line1: first line: base64(*nonce*)
- line2: second line: base64(*response cipher-text*)

3.3.2. 'upload' command

Rebecca expects the following JSON fields:

```
{
cmd:       'upload'          # well known string
to:        h2(bob_comm_pk)  # hpk of communication destination (Bob)
nonce_id: int[24]          # Alice's random nonce for Bob, which also ser
ves as a unique communication ID
payload:  int[..]          # a binary blob (up to 100kb), presumed to be
a cipher text between Alice and Bob encrypted with nonce_id, and whatever
 keypair Alice/Bob ratchet agreement currently dictates. Clients can use
(a_comm_sk, bob_comm_pk) at the cost of forward secrecy.
```

Rebecca will cache *nonce_id, payload* into memory keyed as the next communication for Bob's hpk_bob = h_2(bob_comm_pk) with a 3 day expiration (config param)

Rebecca doesn't know and can not verify if the *hpk_b* value in *to* corresponds to any real public key or not. It only holds the communication queue for any well formatted hash value.

Rebecca doesn't provide Alice with any verification of communication delivery. The communication will be erased from memory in 3 days (config param). It is up to Alice and Bob to confirm the mutual communication delivery via an internal application protocol.

3.3.3. 'download' command

Relay expects the following JSON fields

```
cmd:   'download'      # well-known string
from:  [hpk1,hpk2,...] # optional. A list of comma separated *hpk*s of co
mmunication originators to filter. If omitted, downloads all communicatio
ns.
start: 10              # optional. The first communication position, defa
ults to 0 if omitted.
count: 50              # optional. The total number of communications to
download, defaults to 100 if omitted.
```

Rebecca encrypts with (*relay_nonce,a_sess_pk,r_sess_sk*) the JSON array of communications sorted by the order in which they were received

```
[{
  received_at:  # UNIX UTC timestamp of the communication receipt
  from:         # hpk of Bob's comm key
  nonce_id:     # communication nonce received from  Bob and the unique c
ommunication ID
  payload:      # cipher-text received from Bob
},…,…,…]
```

Rebecca populates the array until the one of the following conditions are met:
- The array has either count or 100 communications (100 is a config param)
- The array has all communications for the given *hpk*

Rebecca's response:

- line1: first line: base64(*relay_nonce*)
- line2: second line: base64(*communications array cipher text*)

Alice will check time-stamped nonce and decipher the array with (*relay_nonce, r_sess_pk,a_sess_sk*)

Using the *from* field allows Alice to do whitelisting if for whatever reason her traffic is overloaded. Alice can download and process communications from her whitelisted collection of *hpk* before downloading the rest of her communications from unknown parties.

3.3.4. 'delete' command

Rebecca expects the following JSON fields:

```
  cmd: 'delete'        # well known string
  ids: [id1,id2,...]  # A JSON array of communication nonce_ids to be del
eted
```

After the communications are successfully deleted, Rebecca responds with a plaintext *count* of communications for *hpk* in the response.

4. Reference Implementations

- Relay node: Rails/Ruby, https://github.com/vault12/zax
- Relay client: CoffeeScript, https://github.com/vault12/glow
- Test Relay: https://zax_test.vault12.com/
- Test dashboard: Angular/Coffee, https://github.com/vault12/zax-dash
- Questions or Comments? Our community Slack: https://slack.vault12.io/

---

1. http://arstechnica.com/business/2012/05/skype-replaces-p2p-supernodes-with-linux-boxes-hosted-by-microsoft/ ↩
2. Around 2 billion smartphones as of this writing, estimated to increase to 4 billion within the next few years: http://ben-evans.com/benedictevans/2015/5/13/the-smartphone-and-the-sun. ↩
3. We use Redis for memory storage in our implementation. ↩
4. In the future, it is concivable to leveraging zero-knowledge proofs to eliminate disclosure of long term public keys (and therefore device identity) to a relay node completely. ↩
5. Such as Axolotl or simular schemas: https://github.com/trevp/axolotl/wiki. ↩
6. NaCl: Networking and Cryptography library: http://nacl.cr.yp.to/ ↩
7. To Hash or Not to Hash Again? (In)differentiability Results for H2 and HMAC http://cs.nyu.edu/~dodis/ps/h-of-h.pdf ↩
8. For relays with SSD storage, temporal caching of the encrypted communication data on hard disks should be an acceptable choice due to the technical challenges of recovering data from properly erased SSDs ↩
9. This is the only place in the protocol where Alice discloses her long term public key. A reliable relay will delete it, while a relay taken over by another agency might keep it and therefore get meta-data about Alice's communication patterns. Eventually, we can leverage zero-knowledge proofs to let Alice prove ownership of *a_comm_sk* without disclosing *a_comm_pk*, while disclosing only the proof for h_2(*a_comm_pk*) ↩