DIMA PROJECT OFFICIAL DOCUMENTATION

"Watchdog"

Claudio Rizzo
800471

Emanuele Uliana
799256

4/7/2014

Teacher: Prof. Luciano Baresi

Version 1.0

Contents

1 Project context and purpose

1.1 Context

1.2 Purpose

2 Project planning

2.1 Time schedule

3 Requirements analysis

3.1 Actors

3.2 Functional requirements

3.2.1 Mobile phones association

3.2.2 Mobile phone remote localization

3.2.3 Mobile phone remote mark

3.2.4 Mobile phone remote alarm triggering

3.3 Non-functional requirements

3.3.1 Privacy and security: problems and solutions

The remote control of a cellphone is a critical activity and has many security and privacy requirements: the next paragraphs show them briefly: for in-depth explainations see the design section (4.3).

Sender authentication

While the sender (telephone) authentication plays indeed a key role, it's even more crucial the authentication of the person behind a control message; that's the reason for employing a password based authentication scheme: in the initialization wizard the user is required to insert a password which is going to be needed to send a message to that telephone (the basic assumption is the password is known only by the mobile owner and by some people, possibly no one, he trusts). The password is stored hashed with SHA-256 in the application preferences, along with the hashing salt (a random token) to avoid both time-to-memory attacks (such as rainbow tables) and the equality of two hashes generated from two equal passwords; the salt is sent to another telephone after the process of public keys authentication (See section 4.3.6).

Message integrity/authentication/non forgeability/non repudiation

The command messages have some specific security requirements (plus confidentiality which is explained in the next paragraph:

Integrity

The message received must be exactly the one sent: every transmission error or tampering must be detected and cause the abort of the current command session: no retransmission is done.

Authentication
The receiver must have a secure way to understand which telephone the received message comes from.

Non forgeability
Nobody should be able to forge a command message which is both valid and correctly authenticated.

Non repudiation
The sender must not be able to deny he sent a specific message (if he actually did it).

Digitally signing every command message can ensure integrity, authentication, non repudiation and a weak defense against non forgeability: symmetric encryption (and in particular AES-256 in GCM mode of operation) is needed for full protection.

Message confidentiality
No one should be able to detect that and which command is sent to a mobile phone, so the command message is encrypted with the symmetric cipher AES-256 in GCM mode of operation (used for performance reasons and for a supplementary integrity check).

Asymmetric keys management
Digital signatures (and shared secrets computation as we will see) require asymmetric cryptography: in the initialization wizard the application generates and stores in the preferences a key pair based on the elliptic curves; the reasons for this choice are performances and the smaller key length with respect to other keys (like RSA and DSA ones) at a fixed level of security. This makes the 140 characters (bytes) Android limit for a single sms no more a problem.

Symmetric key/initialization vector management
AES-256, being a symmetric cipher, encrypts and decrypts a specific message with the same key, and, given the communication channel is not secure, the two parts must agree on the same key in some way; in particular ECDH is used to compute a common secret once and for all, then, when in need to send a message, the sender picks up a random 32 bytes salt, forwards it to the receiver, then both parts use a keyschedule algorithm (PBKDF2 with HMAC-SHA-256) to derive the same key starting from the secret and the salt. Furthermore the GCM mode of operation requires for every message the sender to generate a 12 bytes random initialization vector and to send it to the receiver.

Public keys mutual authentication
While dealing with asymmetric cryptography, the main problem is to bind a public key with a real user to avoid active Man-In-The-Middle (MITM from now on) attacks. Neither a Pulic Key infrastructure (PKI) or a Web Of Trust (WOT) is employed, because they are both potentially insecure for various reasons (in the PKI case the presence of a trusted element, a certification autorithy hierarchy, which may be compromised/untrusted/fake; in the WOT case the presence of a net of trusted elements, the ones who signed a specific public key, which might be fake/bad persons; furthermore a key with no signatures is not automatically a fake one, but there isn't a way to tell), so the application uses a modified version of the Socialist Millionaire Protocol (SMP) to authenticate to each one each other key; this requires the two parts to have a common secret (an answer to a particular question set up on the fly by the users during the SMP), which is easy to achieve, since the two users are likely to be the same person or two people who trust themselves.

3.3.2 Human friendly interface and transparency

3.3.3 Performances

We chose the crypto algorithms with an eye on the performances of the whole system: the key idea is the bottleneck must be the sms and not the computation time required by the encryptions/decryptions; for this reason the command messages are encrypted with a symmetric algorithm and not with RSA or ElGamal (or another asymmetric algorithm), since symmetric cryptography is faster than asymmetric at least by two orders of magnitude (they are very likely to be 3 anyway); however to do ECDH and ECDSA the application needs also an asymmetric key pair, which is generate during the intitial wizard once an for all, so an acceptable overhead. The public keys mutual validation (SMP + ECDH in practice) takes some time, but it's done only one time per association, which means two telephones have to do it only when they associate themselves. Finally the digital signature/verification process are quite fast and so is the key-derivation from the secret and the salt.

3.4 Use cases

3.4.1 Initialization wizard

3.4.2 Mobile phones association

3.4.3 Remote control: localization

3.4.4 Remote control: mark stolen/lost/both/found

3.4.5 Remote control: alarm triggering/untriggering

4 Design

4.1 Application Architecture

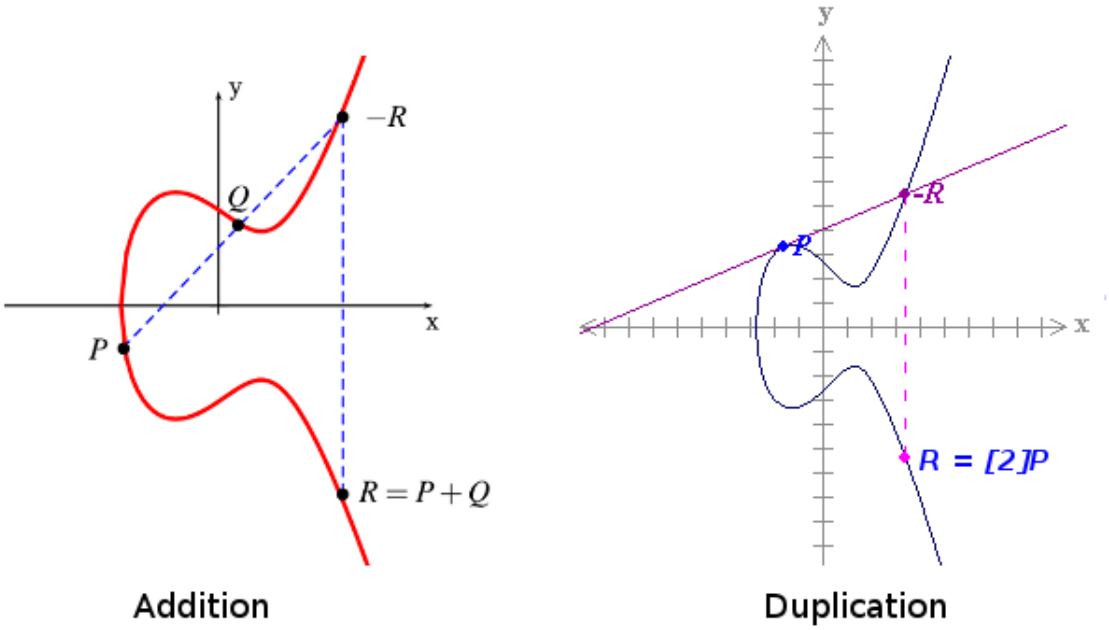
4.2 Design Patterns

4.3 Crypto protocols and alogrithms

The application makes a heavy use of cryptography, so we needed a good crypto provider for java, which we believed to have found in Bouncycastle; however its libraries are not convertible into the Dalvik format, so we had to rely on Spongycastle, an unofficial Bouncycastle porting for Android. Unluckily some algorithms/protocols we had intention to use (namely FHMQV/ECMQV and the native SMP) are not supported (no java implementation for them found), so we ended up using ECDH and a homemade version of SMP instead.

4.3.1 Elliptic Curves key pair generation

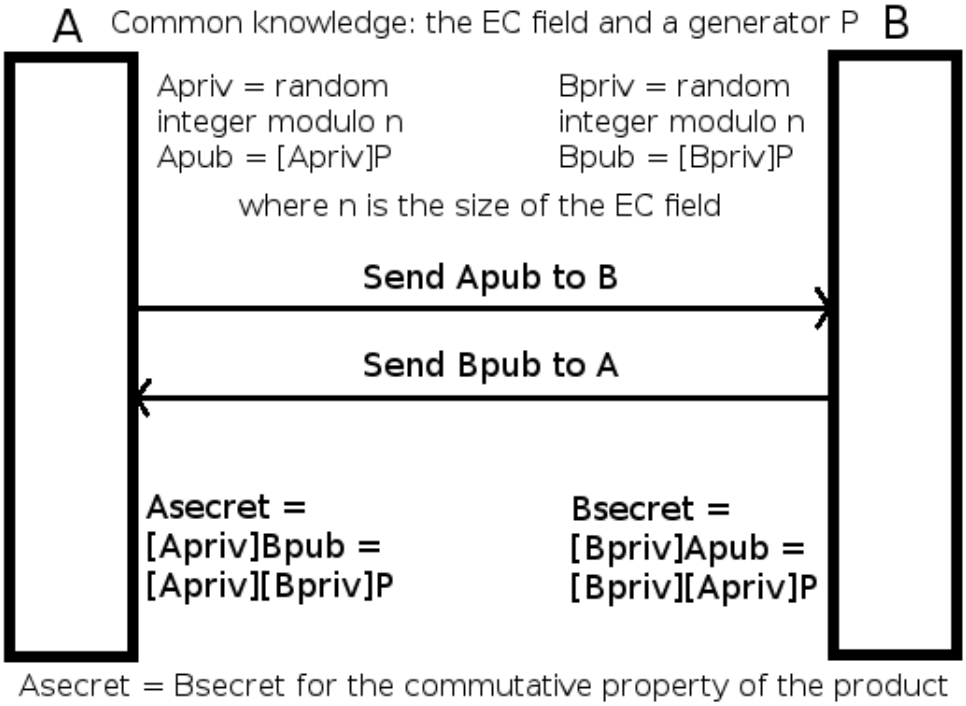
The public/private key pair generation is done by using elliptic curves for performance and memory complexity reasons (at a fixed security level the EC keys are more than 10 times shorter compared to the RSA/DSA/ElGamal ones): a message in Android has a maximum length of 140 characters (bytes), so we had no choice if we didn't (and we didn't since it would have screwed up our crypto layer) want to use multipart messages. The curve used is a NIST standard: "secp256r1", also known as "prime256v1", which generates 256 bits long keys (we actually thought to use "secp521r1" for enhanced security, but the keys were too long). Using a named curve which is also a standard has a few advantages: first, it generates automatically all the parameters needed, second, its security has been widely tested by the cryptanalists of all the world. The generated keys are encoded into byte arrays in different ways: the private ones using the "PKCS8 encoded key specifiers", the public ones with the "X509 encoded key specifiers". It's always possible with a key factory to decode both encodings leading to keys identical to the pre-encoding ones. The keys are generated this way: given the order of the EC group and a randomly chosen point of the curve (different from the one at infinity), which is also a generator of the group, the private key is that point and the public key is computed as [priv]P, where the multiplication denoted by [integer]Point is reduced to a sequence of application of doubling a point and addition between two points. The pictures below shows graphically how these operations are defined for elliptic curves.



Picture 1: ECDH schema

4.3.2 Elliptic Curves Diffie Hellman key exchange

The Diffie-Hellman key exchange (DH) is a key-agreement protocol used to generate a common secret between the two parts over an insecure channel; the computed secret is guaranteed to be the same for both and it's in the form of a byte array (in our case with length 32). From that secret then a deterministic key generation algorithm is able to extract a symmetric key usable for encryption/decryption. The Elliptic Curves DH (ECDH) works like this: every part computes a key pair over the same EC (we use the ones computed in the wizard), then both send their own public key to the other; now they multiply (for a proper definition on multiplication in a EC field) the public key of the other by their own private key: the result is the same due to the public/private EC keys mathematical properties as shown in the picture below.



Picture 1: ECDH schema

The problem with ECDH is the lack of authentication of the received public key: an active MITM could impersonate user A with user B and user B with user A by tricking them into believe his public key belongs to the other side while actually it's not true. This single point of failure is solved by embedding ECDH into SMP (see section 4.3.6).

4.3.3 Elliptic Curves Digital Signature Algorithm

Nowadays ECDSA is the best known algorithm for computing digital signature with a reasonable size, high performances and very high security: it's the EC variant of the DSS-DSA algorithm and it works like this: assumed the signer (sender) has a keypair based on EC (Apriv = s, Apub = [s]P), and the receiver knows Apub and trusts it, and both parts know the curve and its parameters (n = size of the group, P a generator of the group), then:

1. The signer chooses a random integer $r \ mod \ n$ such that $n > 0$ and $GCD(r, n) = 1$
2. The signer computes $[r]P = (x, y)$
3. If $x = 0$ goto step 1, else the signer stores x as k
4. The signer computes $r^{-1} \ mod \ n$ and $e = SHA - 1(m)$ where m is the message to sign
5. The signer computes $z = r^{-1}(e + sk) \ mod \ n$
6. If $z = 0$ goto step 1, else the signature is made by (k, z)

Verification:

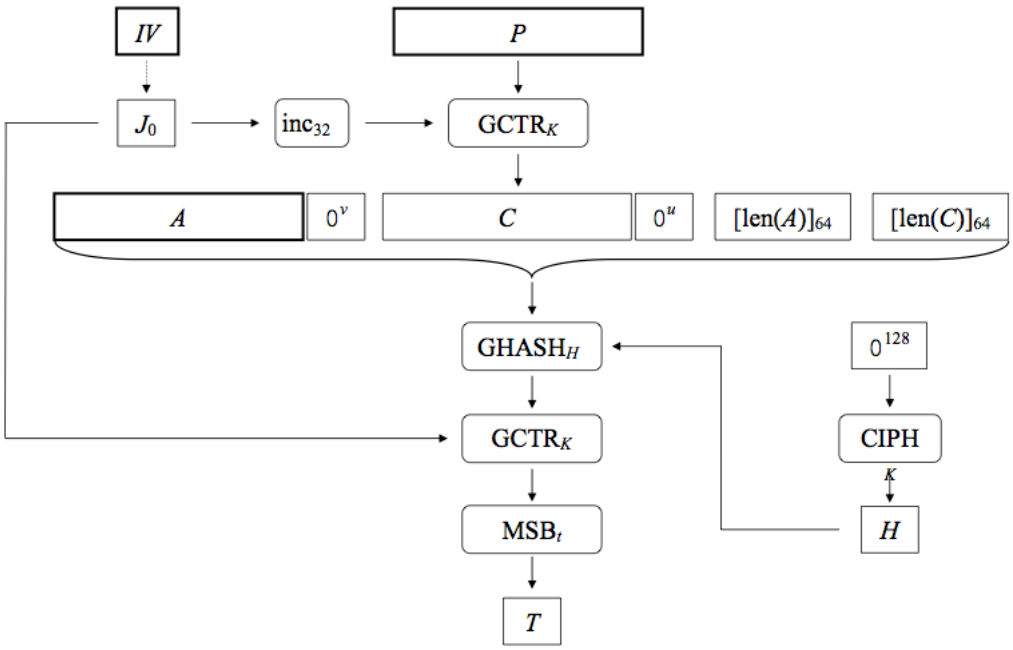
1. The receiver checks whether k and z are $mod \ n$ and positive not null integers, if not, the signature is not valid
2. The receiver computes $e = SHA - 1(m)$ and $w = z^{-1} \ mod \ n$
3. The receiver computes $u_1 = ew \ mod \ n$ and $u_2 = kw \ mod \ n$
4. The receiver computes a point $X = [u_1]P + [u_2][s]P$ and stores it as (x, y)
5. If X is the point at infinity or is not verified $x = k \ mod \ n$, the signature is not valid, otherwise is valid.

4.3.4 PBKDF2 with HMAC-SHA-256

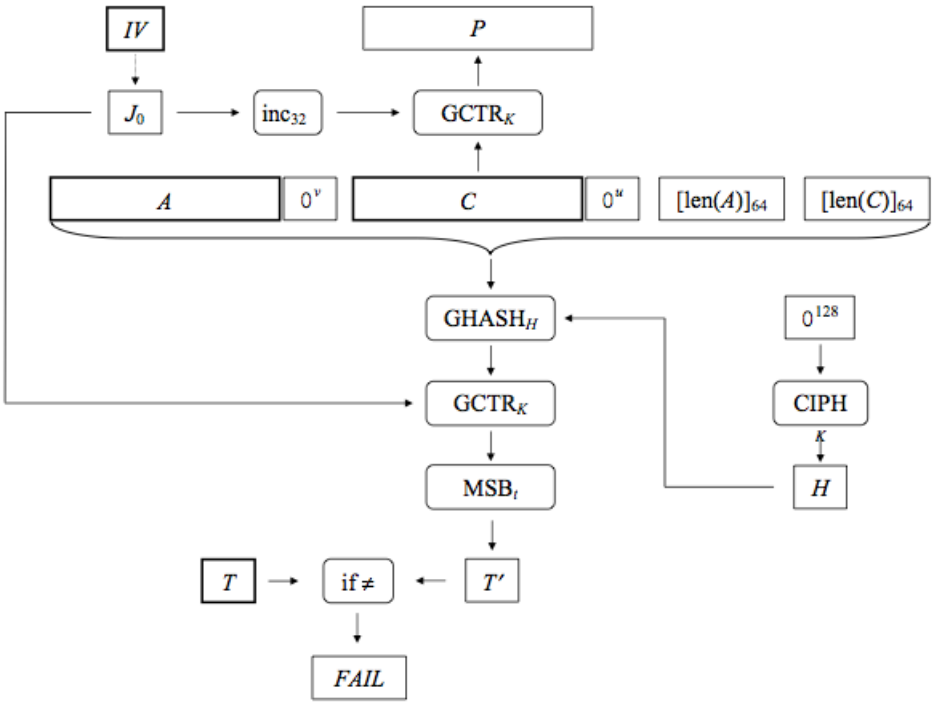
The Password Based Key Derivation Function #2 is a deterministic key-derivation algorithm which generates a symmetric crypto key with a desired length by taking in input a password or a passphrase (in our case the secret from ECDH) and a salt (which makes possible to derivate different keys from the same password) and doing a certain number (4096 in our case) of application of a pseudorandom function (HMAC-SHA-256 in our case) to them in the so called "key stretching", which makes very hard the use of rainbow tables for cryptanalysis.

4.3.5 AES-256-GCM

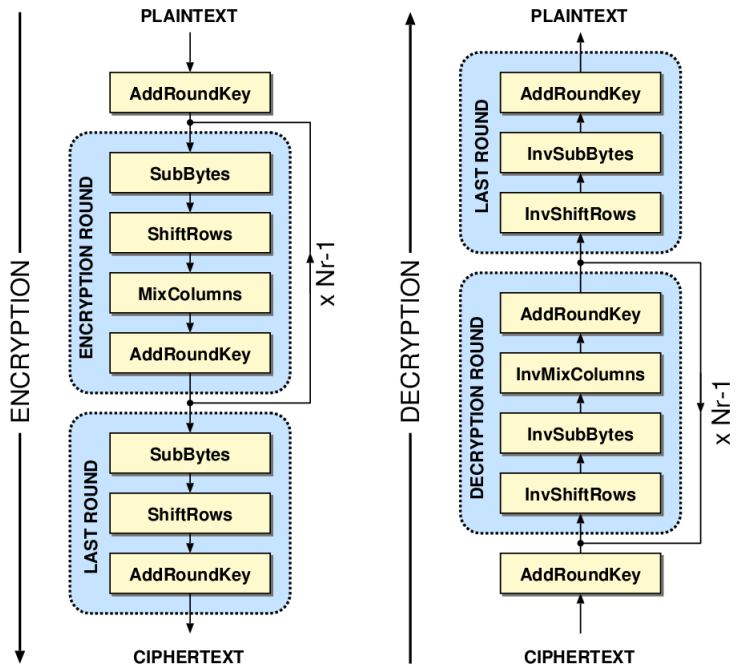
The Advanced Encryption Standard is the symmetric encryption standard algorithm (NIST FIPS-197) almost worldwide since 2001, and it's known for high performances and security against cryptanalysis; in the application we employed its version with a 256 bits key, which has a security margin comparable to RSA with a 15360 bits key. Actually, since it's used in Galois/Counter Mode (GCM), it does not encrypt directly the plaintext, but it's instead used together with a 12 bytes initialization vector IV (which doesn't need to be secret, but unpredictable for every encryption, so the best choice is to pick it up randomly) to generate a pseudorandom keystream, which is then XORed to the plaintext: basically the block cipher simulates a stream cipher (so no padding is needed for the plaintext, whose length, thus, does not need to be a multiple of 16 bytes, like in direct encryption, which is a good thing, considering the 140 bytes limit for Android messages) which simulates a One Time Pad (OTP), the only cipher perfectly secure. Obviously AES-GCM is not perfectly secure because, frist, soon or later, a new key will be equal to one used in the past (not so soon though), second, the key is shorter than the plaintext and, third, the keystream is only pseudorandom. In addition to this the GCM mode generates also a 16 bytes Message Authentication Code (MAC) which is prepended to the ciphertext in order to provide an additional integrity check. Before the decryption (which needs of course the same IV and the same key) the message integrity is verified and, if it's not the case, an exception is raised and the command session is aborted. In GCM mode it's also possible to attach to the ciphertext some non-encrypted data (called Additional Authenticated Data, AAD) which are used along with the ciphertext itself to produce the MAC; in our case this feature is useless, so no AAD. The three pictures below show the encryption and the decryption under GMC (P = plaintext, K = key, C = ciphertext, A = AAD, T = MAC) and the AES encryption/decryption used to generate the keystream.



Picture 1: Encryption in GCM mode of operation



Picture 2: Decryption in GCM mode of operation



Picture 3: Encryption/Decryption in AES

4.3.6 Socialist Millionaire Protocol

The Socialist Millionaire Protocol purpose is to mutual authenticate public keys, so that a logical binding is created between a key and a phisical device; this solves the active MITM problem. Since a java implementation of the original SMP was not available, we coded our own version of it, which also embeds ECDH to spare time: the whole protocol is based on messages exchanging.

Public key request

When someone wants to associate his telephone with another one (and vice versa, which is automatic), and confirms his will by clicking on the apposite button, the system stores the inserted secret question and its answer, then generates a message made by the header "CODE1FFF" (intended hexadecimal as all the similar codes later descripted) and a null body, and sends it to the other mobile, writing into the preferences that the message has been sent. The meaning of the code is a request for the other public key. The receiver, after verifying (in the so called from now on "message validation") the message arrived in a right moment (i.e.: not during another SMP session with the same user), writes into the preferences the occurred event.

Public key sending

The response to a public key request is a message made by the header "CODE2FFF" and the public key itself; of course also the happening of this sending is saved in the preferences updating the SMP log file. The receiver, after the sms validation, writes into the preferences the event and stores in the so called "KeySquare", a keyring of the not yet validated keys, a Base64 representation of the key.

Question sending

After receiving the public key, the system saves the event in the preferences, then fetches from there the previously stored secret question, sends it to the other appended to the header "CODE3FFF" and saves into the preferences this sending event. The receiver validates the message, updates the SMP log file in his preferences and stores the received question.

Hash sending

After storing the secret question, the system notifies the user of the existence of a pending association request: the person at this point can decide to refuse (which is managed as an error) or to insert the answer to the secret question and confirm. If we are in the first half of the SMP (the user who can accept/refuse to send his key has not validated the other public key), also the secret question/answer to be used in the second half must be provided (basically they are stored in the preferences). Now the system computes a SHA-256 hash of the concatenation of the public key and the given answer;

the next step is sending to the other such hash prepended by "CODE4FFF" and to update the SMP log. The receiver validates the message and updates his log too.

Ack and password salt sending

After validating the hash message, the system fetches the key previously stored in the keysquare (removing it from there) and the secret answer saved in the preferences an the beginning of the SMP, and computes an hash in the same way the other telephone did. If the two hashes don't match, the public key is not validated and an error occurs, otherwise the key is saved in the keyring; now the system takes the password salt (saved in the preferences during the initialization wizard) and sends it to the other cellphone prepended by "CODE5FFF", which has the meaning of an ack; then of course the log is updated. In parallel a secret is computed with the received public key and the private key fetched from the preferences: in this sense ECDH is embedded into SMP. The receiver validates the message, updates the log, and stores the salt into his preferences, encoded in Base64.

Second half

After validating the ack message and saving the salt, the system checks whether exists in the keysquare an entry for the other mobile public key. If yes the SMP is successfully over, otherwise a public key request message is sent and the log is updated, starting this way the second half of the SMP. The secret question/answer to be sent to the other are the same ones inserted in the first half, so this time, when the accept/refuse screen is presented, only the secret answer to the other secret question has to be inserted. At the end of the SMP both logs are filled, both public keys are validated, and both systems computed the same 32 bytes common secret.

Error management

Every ecception raised by the system, every message not validated, the hashes mismatch and the user refuse to the association are all treated in the same way: all the preferences relative to the other user are deleted, then a message made by "CODE6FFF" is sent to the other. Such code is accepted by the system if and only if the SMP session is still pending and not finished; the code has the effect of making the system delete all the preferences relative to the other mobile. After that a new "CODE6FFF" is sent; the whole thing doesn't loop because the "CODE6FFF" is ignored if there are no saved preferences relative to the other cellphone. This ensures in case of errors both telephones delete the preferences (and the SMP log) in which is present a reference to the other one, so in an eventual next SMP between the same two mobiles there won't be errors due to already existent preferences.

4.3.7 Command Protocol

The command protocol is the way a command session is managed: when a user wants to send a specific command to a mobile phone, he goes to the right fragment (the one which enables him to select the command he wants), selects the target number, inserts the correct password and confirms. Now the protocol starts with the first message: m1.

First message

The first message (m1) is constructed by concatenating a specific header, the initialization vector which will be needed to decrypt m3 (as it will be used in its encryption, see GCM explaination section), the salt needed to compute the correct decryption key for m3, and a digital signature (the iv and the salt are computed on the fly; the first is stored, the second is used to compute the encryption key for m3, which is also stored). After the message is sent a 120 seconds timeout is started and the command session log is updated.

Second message

When the first message is recived and validated (signature, header) the receiver stores the iv and generates the key starting from the common secret and the salt; then he generates and stores an initialization vector for the encryption/decryption of m4 and generates also a salt used immediately to generate the m4 encryption/decryption key, which is saved in the preferences; now m2 is constructed with those elements in the same way m1 was (header + iv + salt + ecdsa signature), and sent to the other. After that a 120 seconds timeout is started and the command session log is updated.

Third message

When m2 is received before the timeout (if not it will be treated as m1, and obviously will be rejected in the header validation phase), the timeout is stopped, the signature/header validations are done, the iv is stored and the decryption key for m4 computed and saved in the preferences. Now m3 is created in this way: AES256GCM(key-for-m3, iv-for-m3, SHA256(password) + command code + a space character separator + ecdsa signature); after that m3 is sent to the other, the log is updated and the timeout restarted.

Fourth message

When m3 is received within the timeout (which is immediately stopped), the decryption (which includes the MAC integrity check) is performed, followed by the signature verification. After that, various actions are performed depending on the command received and m4 is constructed as AES256GCM(key-for-m4, iv-for-m4, header + fixed length body with padding + ecdsa signature) and sent. Finally the log is updated and the preferences of the session deleted, so that a new session can begin. When m4 is received (within the immediately stopped timeout) and passes the usual decryption/validations steps, different

actions are performed depending on what was the sent command and, after everything is done, the session preferences are deleted on this side too.

Error management

An error in the m1, m2, m3 or m4 decryption/signature validation/header validation is managed simply by ignoring the message, restoring the log to the state it had before receiving such message and restarting the timeout (if one existed). Otherwise, if the error occurs in the execution of the actions required by the m3 command, the construction of m4 is different: AES256GCM(key-for-m4, iv-for-m4, header + fixed length body with padding and error code + ecdsa signature).

Timeout management

When a timeout countdown reaches zero, the command session log in the shared preferences is deleted, so any future command message will be treated as m1 (and discarded if it is a "late" m2 or m3 or m4).

5 Open Issues and TODO list

Our application has still some issues and some incomplete features: the next subsections speak about them.

5.1 Unilateral Uninstallation

If the application is removed (or the preferences deleted) on a telephone, then, after a reinstall, the owner of that mobile phone won't be able to reassociate to his old contacts, since on the other side the situation will still look like a completed association, so any SMP message coming from an already associated phone (including IDontWantToAssociate) will be simply discarded in the message validation phase. A possible workaround is to notificate to all the contacts when the application is uninstalled, but

- it may not be always possible ("dirty" uninstall ways);
- the application is mainly intended to be used by the same person on two telephones, so the user has likely the control on both (he is likely the one to uninstall the app);
- a validation of such message (real or fake or error) is not trivial.

The second possible solution is to create a way to locally delete some preferences relative to the other user(s).

5.2 Association Deletion

Our application lacks a way of deleting a specific association without uninstalling the application (see previous subsection): that's a TODO.

5.3 The Mark Features

The "mark" features (stolen, lost, both, found) are all left unimplemented (despite the application being scalable: the visitor pattern already contains the classes needed to implement them). That's also a TODO.

6 Testing

6.1 Cryptography

6.2 Protocols

7 Installation and usage manual

7.1 Installation

7.2 Usage

7.2.1 Initialization wizard

7.2.2 Change application settings

7.2.3 Send a command message