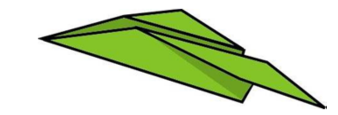
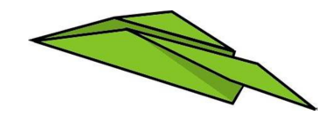
Implementing a verifiable voting protocol: First lessons learned from a proof of concept

Context



- Security requirements regarding 2nd generation e-voting systems in Switzerland have been defined
- They include
  - individual verification,
  - universal verification
  - A cryptographic protocol to achieve these goals
- One such protocol has been described as an example by the sub-working group (UAG) that worked on the requirements
- Providers of 2nd generation e-voting systems will have to implement such a protocol.

The project

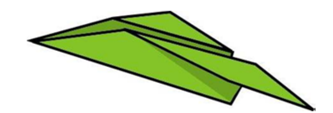


- The goal of our project is to implement the example protocol in order to gain insights in issues and challenges that can arise.

- Today's presentation is about the first issues and challenges we identified so far.

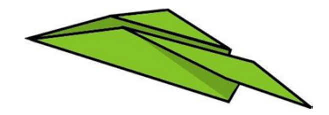
- This project is work done for the Canton of Geneva

Agenda



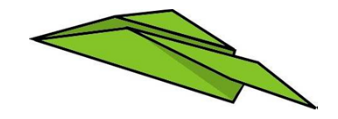
- Specifics about the Technical Regulations
- Brief description of the protocol
- Interesting issues
  - Crypto librairies
  - Performance
  - Representation of codes
  - Information needed by the server
  - Redundancy

A word of caution



- This protocol was given as an example to illustrate security requirements
  - There is no claim that it is correct or compliant
- Our implementation is only partial
  - Mixnets not implemented (more standard)
- This is not a presentation about a perfect protocol or an evoting product

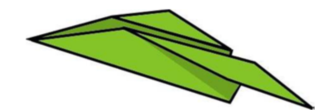
Specifics about Swiss electronic vote



Verifiability

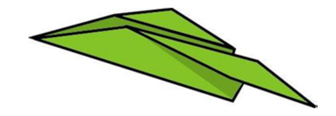
- The goal of verifiability is to be able to give irrefutable proofs
- Even if we have proofs, we need to trust some elements to make falsifying proofs very hard
- The Technical Regulations (TR) define 2 new levels of security and a trust model for each one.

Security levels

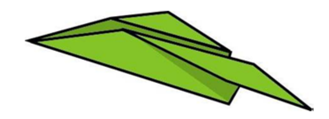


- Individual verifiability:
  - Can be used by 50% of voters of a canton.
  - Voter must be able to detect if vote was not registered as intended.
  - The printer and the voting server are trusted.
  - (dedicated voting devices are trusted)
- Universal verifiability
  - Can be used by 100% of voters.
  - Additionally it must be possible to detect if the result is not correct.
  - The printer and one out of *n* components of the server are trusted.
  - (dedicated voting devices are trusted)

Individual verifiability



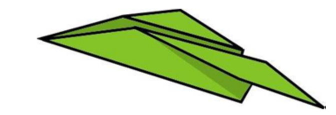
- For example the voter compares
  - A code received by the voting server (trusted)
  - A code printed on a code list by the printer (trusted) and transmitted by the post (trusted)
- According to the TR, chances must be smaller than 1/1000 that a fake code is not recognized by the voter.
- The goal is to detect any manipulations by the voter’s platform or a man-in-the-middle.



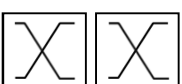
Universal verifiability

- Proof that the result corresponds to all received ballots
- Can be direct, as in bulletin boards
  - Difficult to achieve while preventing vote buying
- Can be indirect: verification is delegated to auditors who can access the proofs

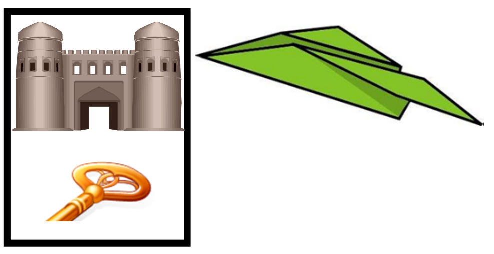
The protocol



Concepts

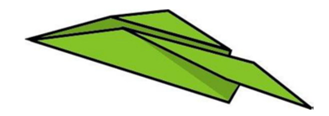
- Based on homomorphic encryption, El Gamal
- Return codes are generated for individual verifiability
- Zero knowledge proofs
  - Proof of no alteration during mixing, proof of correct calculation of return code, proof of attribution of a vote to a vote card.
- Distributed keys
  - Nobody knows the complete private key
- Control components store and manipulate the private key
- Verifiable reencrypting mixnets 

Control components



- Protect their part of the private key(s). Create a signed log of all operations done with the key.
- Execute very simple mathematical operations with the keys.
- It is the only element that needs to be secure in the VE system.
- Only one of the control components needs to be honest to detect any manipulation of the result or any violation of confidentiality.

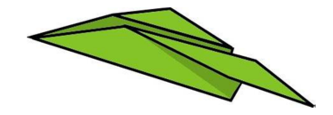
Phases of the protocol



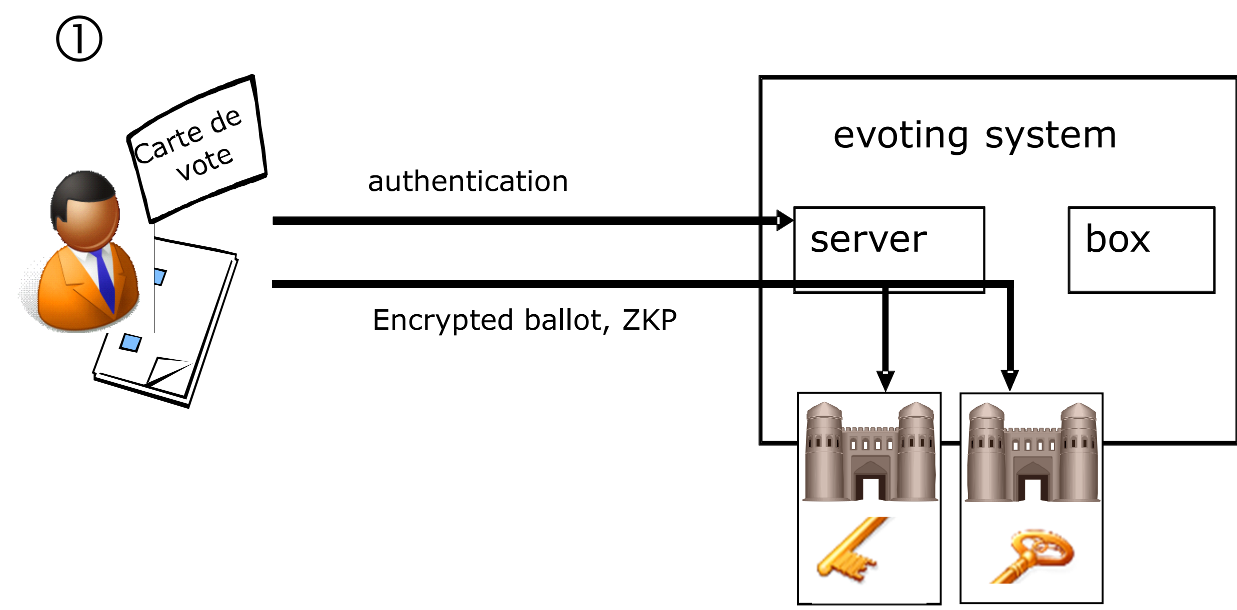
a) Preparation: generation of the keys, of the voting card and the code lists

b) Vote: encryption of the vote, calculation of the return code, storing in the ballot box

c) Counting: mixing the ballots, decrypting the ballots.

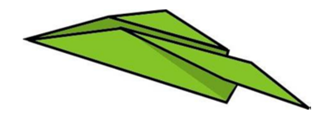


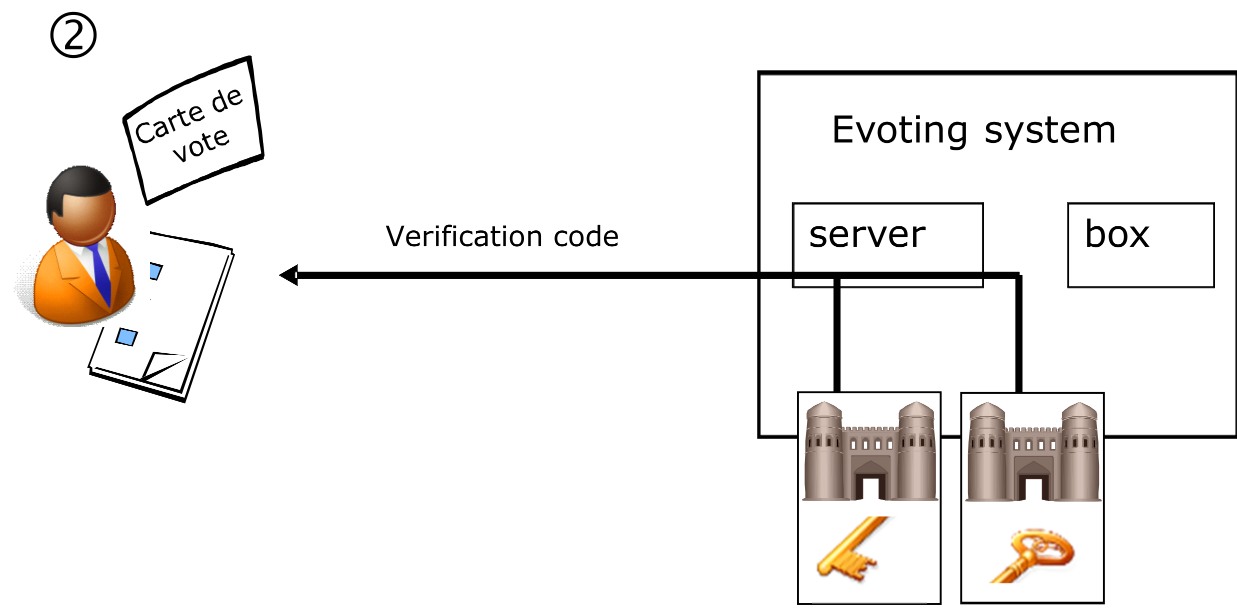
b) vote 1/4



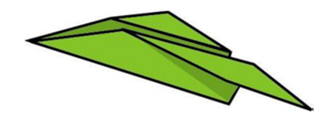
- The voter authenticates: voting card number **n,** code list **id,** other elements
- Choses a candidate(s). Sends encrypted **candidate code** plus ZKP proving knowledge of vote and link to card number **n**.

b) vote 2/4

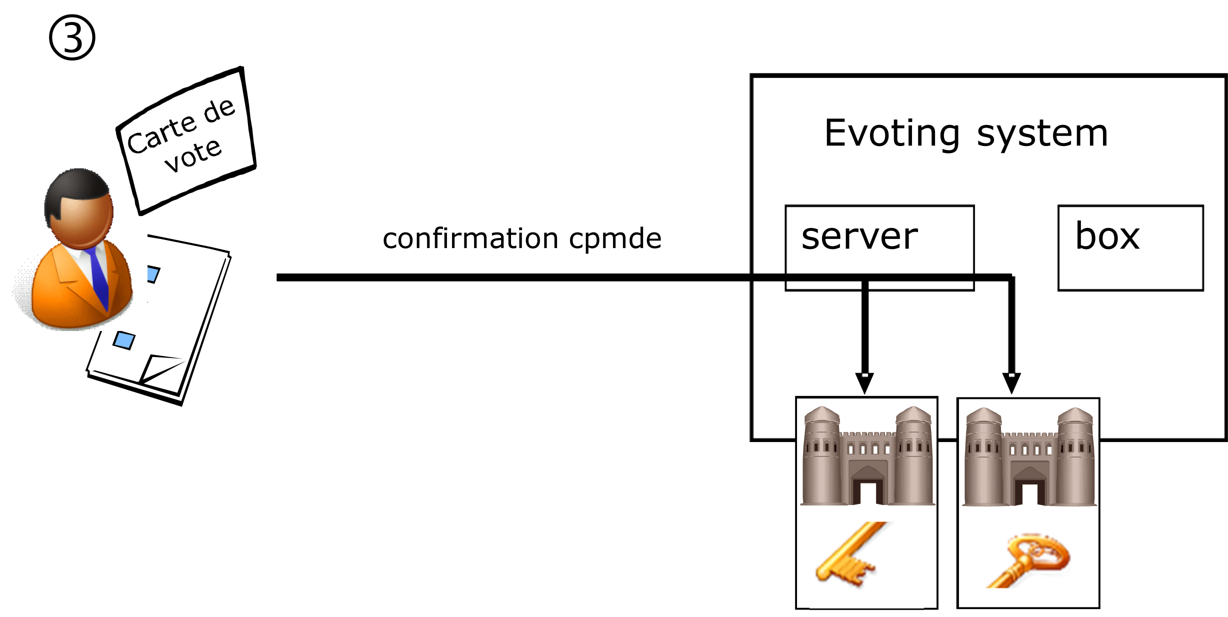




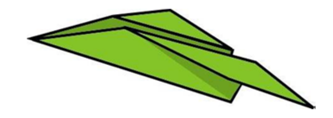
- The server lets the components blind and then decrypt the vote. ZKPs are generated for both operations.



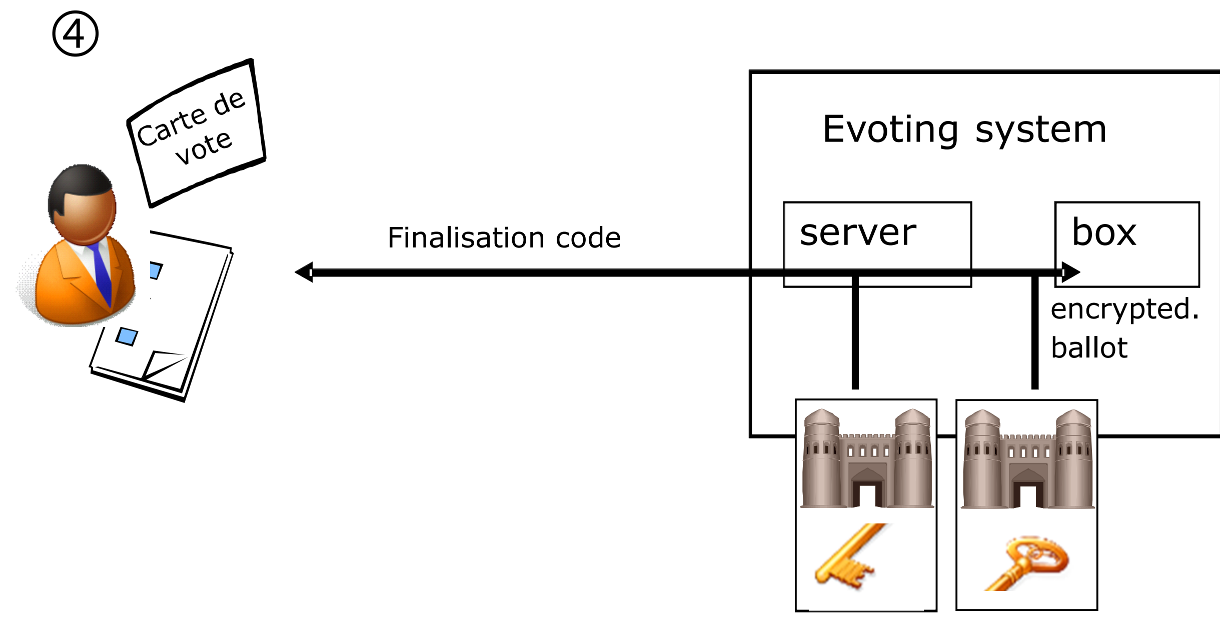
b) vote 3/4



- Voter verifies the verification code and sends **confirmation code**.

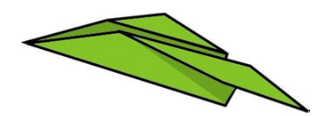


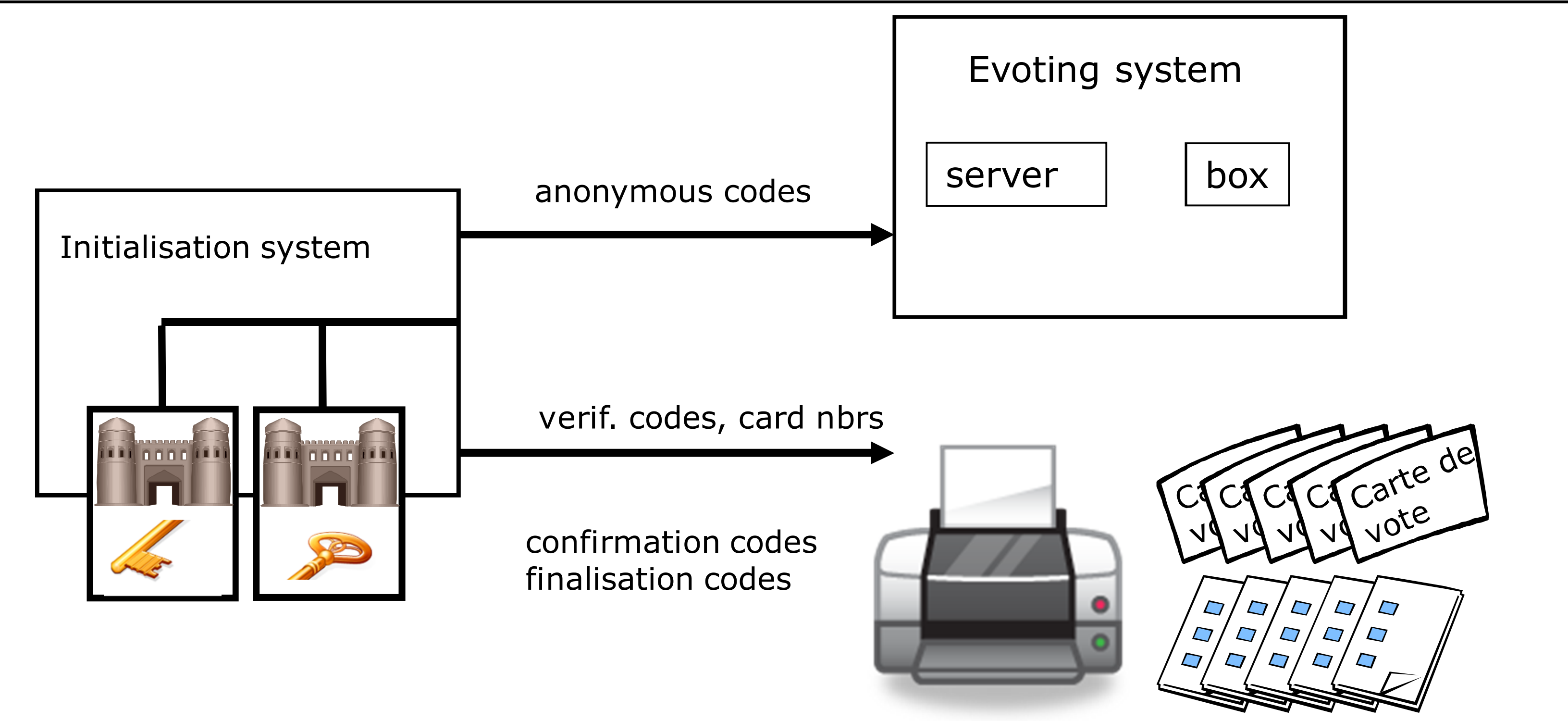
b) vote 4/4



- If the confirmation code is correct, the system drops the vote in the box and the control components generate the **finalisation code**.

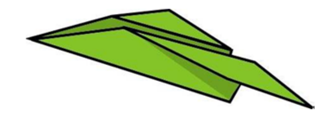
a) preparation





listes de codes

- Control components generate private key
- They generate verification codes encrypted for the printer
- They generate and mix verification codes for the evoting system.



c) counting

counting system

mix

mix

decrypt

box

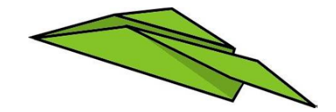
server

results

proofs

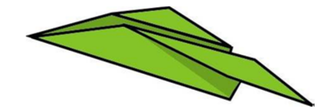
Findings

Crypto libraries



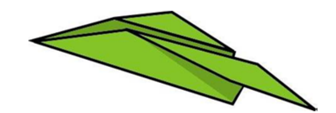
- We do ElGamal encryption
- Existing libraries
  - Java: BouncyCastle, Cryptix, FlexiProvider, Qilin, Verificatum
  - C++: libgcrypt, Botan, Crypto++
  - Python: Pycrypto, eyPyCrypto, Pysecret, Viff
  - Javascript: Forge, SJCL

Crypto libraries



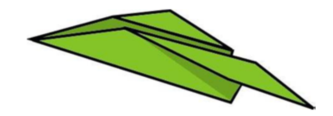
- We do ElGamal encryption
  - But we cut the key in pieces and do only partial encryption/decryption
  - We want to know or to set the randomness r
- We do standard ZKP (knowledge of log, equality of log)
  - But sometimes we add a card number into the hash
  - And sometimes we want to prove the inverse of the log instead of the equality!
- No standard library does exactly that
  - Except for Unicrypt !

Crypto libraries



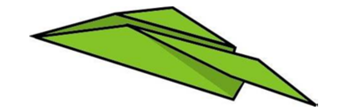
- There are only few operation needed
- Easy to implement them using any standard crypto library
  - We had to define 22 primitives
    e.g generate_privkey, generate_pubkey, zkp, validate zkp, reencrypt, blind, hash, sign, …
- Current HSMs on the market do not have these primitives
  - There are programmable HSMs with crypto libraries

Crypto libraries



- If parts of the systems are implemented in different languages, it is important to define the conversion of data types
- E.g for the ZKP of the vote we need to
  - introduce the card number into a hash
  - Use the result of the hash as a number
- We need to define a language independent way of doing it (long_to_bytes, bytes_to_long)

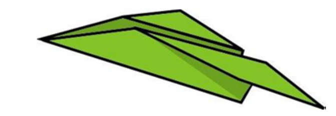
Crypto libraries



- Our choices

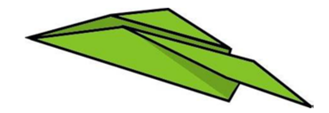
Primitive nécessaire	Java	Python	JavaScript
Support des grands nombres	java.math.BigInteger	inclus	Forge (jsbn)
PRNG	java.security.SecureRandom	pycrypto	Forge
Test nombre premier	BouncyCastle	pycrypto	-
Modexp	inclus	inclus	Forge
Hachage SHA-1	java.security.MessageDigest	pycrypto	Forge
Signature DSA	BouncyCastle	pycrypto	-

Performance



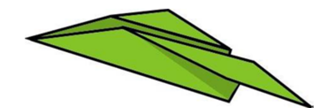
- 98% of the time is spent doing modular exponentiation (modexp)
- Typical speed for 2048 bit modexp on one CPU core (Corie i7 L620)
  - Java 30/s
  - Python 25/s
  - Javascript (Chrome V8 engine) 5/s
- Java and Python use fast C code to perform them. Implementing the protocol in C should not improve much the speed.
- (HSM, 2010: 1000 DSA sigs with p=2048 and q=224)

Performance: example



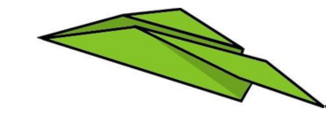
- 250’000 voters
- 1 control component (parallel operation of all components)
- 600 candidate codes (300 candidates because cumulation, plus codes for lists and for blanks)
- 125’000 received ballots with 100 votes each

Performance server-side



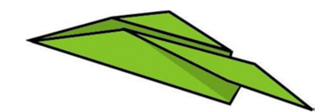
- Operations
  - Setup 3 billion modexp
  - Receiving the ballots: 300 million modexp
  - Counting: 100 million modexp
- To be able to do the setup in one day we would need 1300 CPU cores…
- Cost of mixing has not yet been modeled
  - Could be very expensive at setup time

Performance client-side



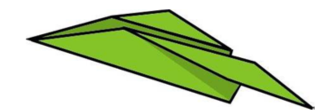
- 5 modexp per vote (2 for encryption, 3 for ZKP).
  - About one second
- If the voter selects 100 candidates we need one second at each selection (ok)
- If the voter selects complete list we could use a single code for the list
- If he selects a list and then removes one candidate…

Possible optimizations



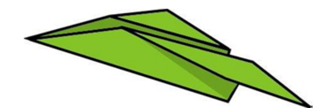
- Change parameters
  - Use a smaller q for G_q in $\mathbb{Z}_p^*$
  - q of size 256 bits is enough to protect against brute-force
  - Eight times faster than q of size 2048 bits.

Possible optimizations

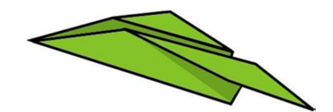


- Use a limited number of code lists
  - Using 10’000 different lists rather than 250’000 reduces setup time by a factor of 25
  - The attacker needs to know at least 10 lists to have one chance in 1’000 to have the correct code
- If the control components know which list has been given with which vote card there is no need to print the id on the list
  - The voter does not need to type the list id when voting
  - The attacker can not now which list is in the hand of the voter
  - Using 1’000 lists should be sufficient (gain is 250)
- Alternatively have 250’000 different list ids but only 1’000 different lists

Possible optimizations



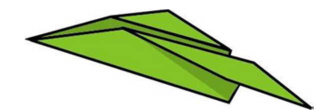
- Let the printer do the anonymization of the codes
  - Instead of recalculating and mixing them for anonymization, let the printer decrypt the codes, sort them, and send them back
  - reduces work by factor 2.5
- Together we could potentially have a gain of 500 - 5000



Representation of codes

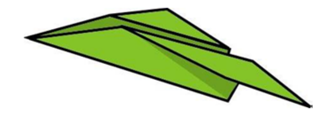
- The verification codes are calculated as the blinded candidate code
  - The voter encrypts the candidate code
  - The control components blind the code with the exponent that corresponds to the code list
  - The control components then decrypt and return the blinded code

Representation of codes



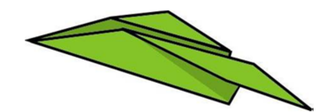
- The blinded verification code is 2048 bits long!
- We need P < 1/1000 chance for the attacker
  - Three digits or two alphanumerical characters could be enough
- If we truncate the code, chances are that two candidates have the same code
  - Technically this is not a problem, P is still 1/1000
  - Voters will not accept to have the same code for two candidates

Representation of codes



- In the example protocol we sort the blinded candidate codes and return the position of the code in the sorted list of the code
- Elegant, the codes run from $1..r$
- If there is less than 1000 codes, then P is too large

Representation

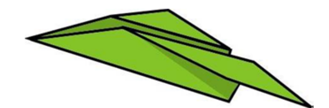


- For elections of parliaments
  - We need two codes per candidate (for cumulation)
  - We need one code per list
  - We need one code per empty line on the lists
  - With 500 candidates we can reach 1000 codes
- For referendums or small elections
  - We could truncate the blinded candidate codes to 4 digits and reject lists that contain collisions

Information shared with the server

- The server helps
  - Authenticating the user
  - Verifying the verification code
  - Receiving the confirmation code
  - Returning the confirmation code
- The codes are short (e.g. not 128 bits, bruteforceable)
- We must be careful that the server does not learn enough to impersonate a voter or the server.
- At some point the control components probably need to have a way to make a decision among themselves

Redundancy



- Critical on-line services often run in two redundant locations.
- Control components hold one part of a private key
  - This could be solved by using threshold crypto systems
- They also generate proofs and logs
  - These should be replicated to avoid loss
- They also hold state (e.g. who has voted)
  - This information can also be held by the (untrusted) server
  - Manipulations by the server would be detected afterwards by consulting the logs