Loria
Lorraine research laboratory in computer science and its applications

Inria

UNIVERSITÉ DE LORRAINE

FRANCE 2030

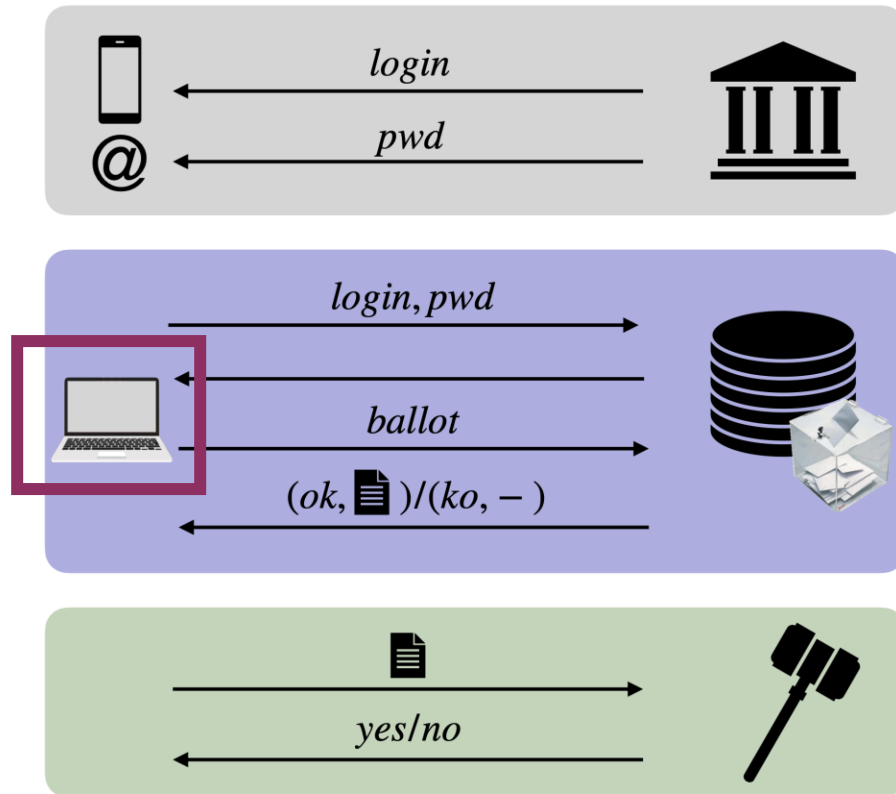
Inria Centre at Université de Lorraine
Loria D2. Formal Methods
Group PESTO

ProVerif proof of an Online Voting System with Short Voting Codes

Florian Moser
PhD student supervised by V. Cortier & A. Debant

05.04.2024

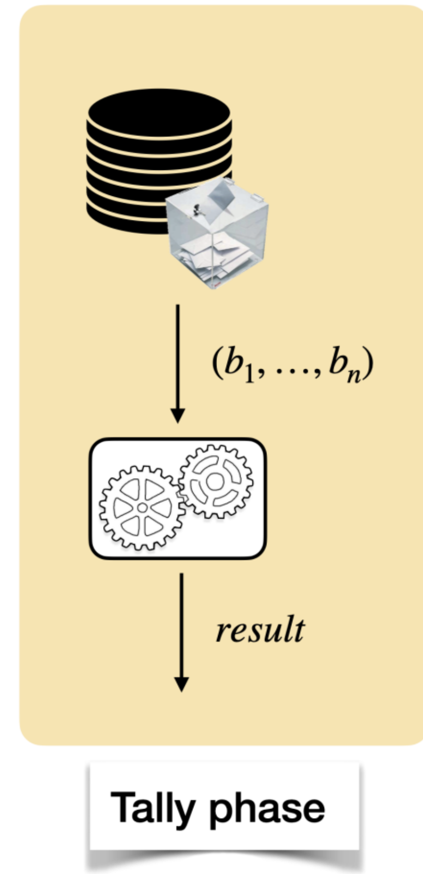
An Online Voting System



Setup phase

Voting phase

Verification phase



Tally phase

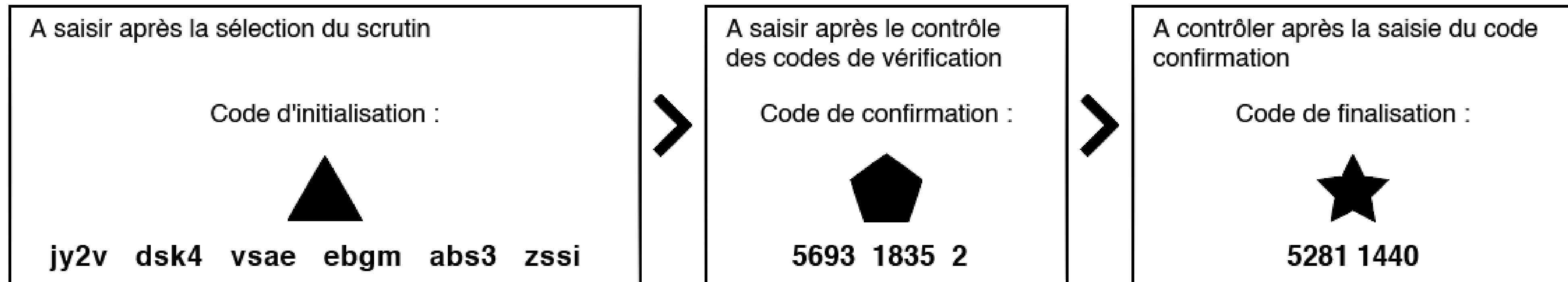
[slide borrowed from Alexandre Debant.]

The Swiss Setting

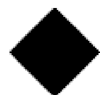
Vote électronique

URL : **https://demo.evoting.ch**

Fingerprint (SHA-256) : b9 ad 65 9d a1 71 e8 a9 4e d1 96 a7 7d dc d2 e7 f1 b0 62 00 1e 90 28 75 cd a3 8f c3 35 9a 90 4f



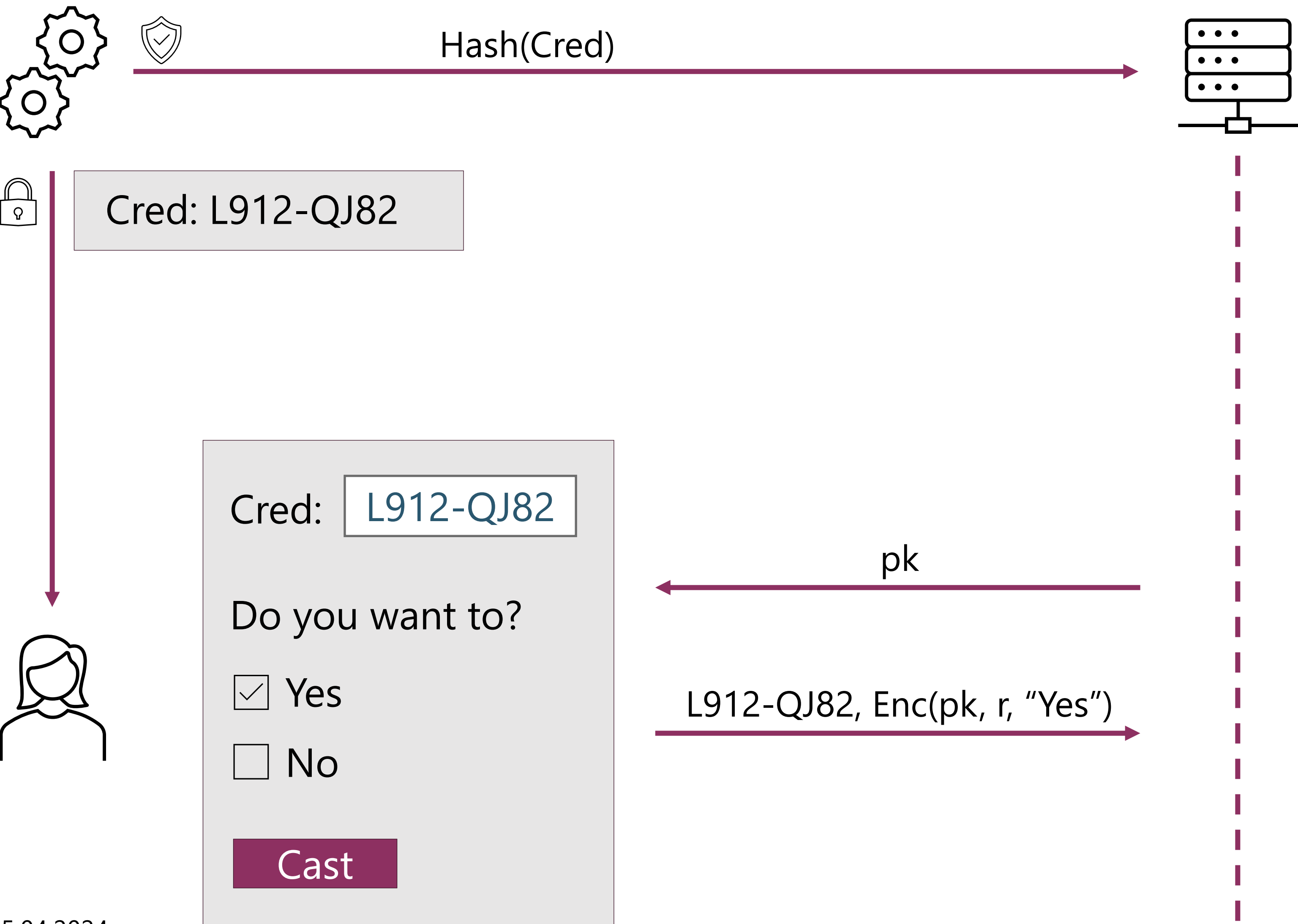
Scrutin du 05.04.2024 - codes de vérification



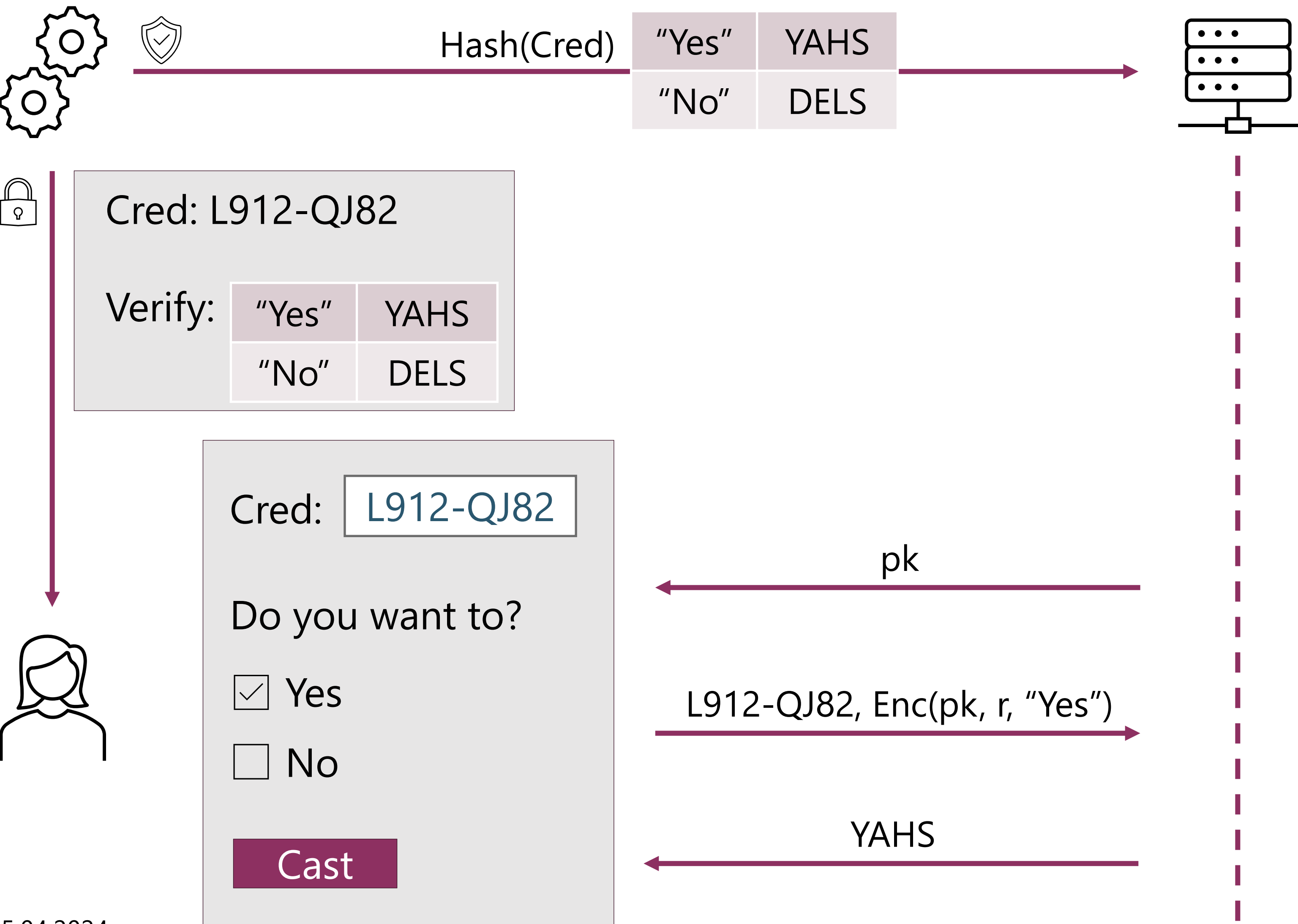
Vous trouverez tous vos codes de vérification ci-dessous

N°	Démo Votation / Demo Abstimmung	Codes de vérification		
		Oui	Non	Blanc
1	Aimez-vous le temps ensoleillé ?	2539	4666	1179
2	Aimez-vous le temps pluvieux ?	8560	1397	0974

A simple voting system

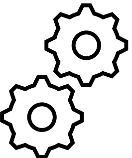


A simple voting system with Return Codes



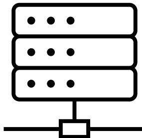
Trust	Veri.	Priv.
Setup	x	x
Server	x	x
Device		x
Internet		(TLS)

A simple voting system with Return Codes and Short Voting Codes



Hash(Cred)

“Yes”	2	YAHS
“No”	1	DELS



Trust	Veri.	Priv.
Setup	x	x
Server	x	x
Device		
Internet		

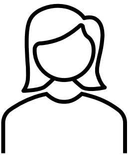
Cred: L912-QJ82

Vote:

“Yes”	2
“No”	1

Verify:

“Yes”	YAHS
“No”	DELS



Cred: L912-QJ82

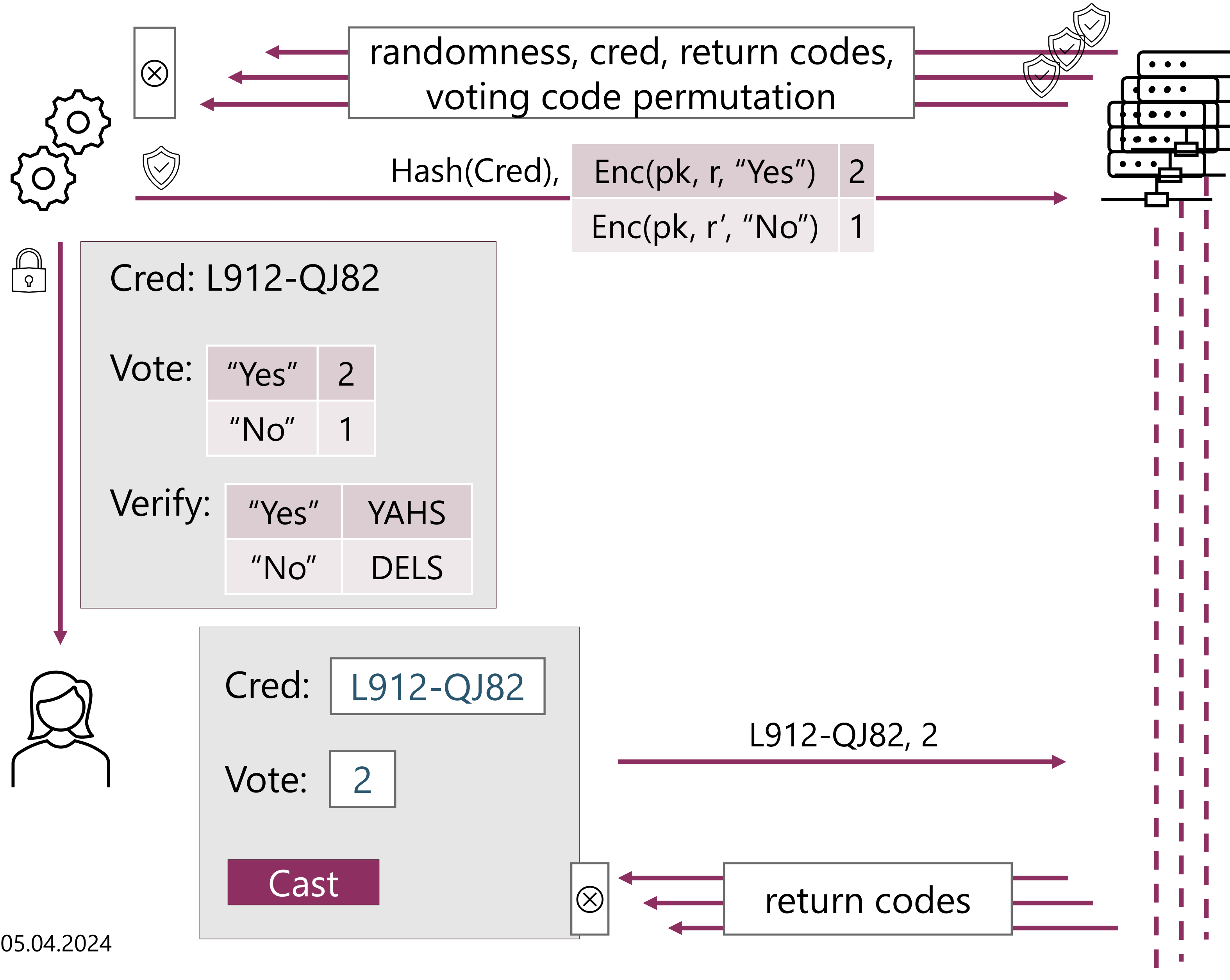
Vote: 2

Cast

L912-QJ82, 2

YAHS

A distributed voting system with Return Codes and Short Voting Codes

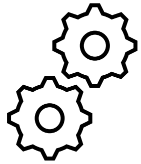


Trust	Veri.	Priv.
Setup*	x	x
Server	1/n	1/n
Device		
Internet		

* Deterministic

Generate distributed deterministic secrets

$$\mathcal{C} = \{1, 2\}$$
$$PtC = [(\text{'Yes'}, 1), (\text{'No'}, 2)]$$
$$\mathbb{P}_{|\mathcal{C}|} = \{[1 \to 1, 2 \to 2], [1 \to 2, 2 \to 1]\}$$



Alg.: MergePartialBallots($\forall i \in [1, m].b^{(i)}$)

$\forall i \in [1, m].(Ctvv^{(i)}, ca^{(i)}, p^{(i)}) \leftarrow b^{(i)}$

for $C \in \mathcal{C}$ **do**

  $\forall i \in [1, m].vv^{(i)} \dot{\leftarrow} (C, .) \in Ctvv^{(i)}$

  $CtVV \leftarrow CtVV \cup (C, \oplus_{i=1}^{m} vv^{(i)})$

$CA \leftarrow \oplus_{i=1}^{m} ca^{(i)}$

$PtC \leftarrow PtC * \prod_{i=1}^{m} p^{(i)}$

return $(CtVV, CA, PtC)$

deterministic

$b^{(i)}$

Alg.: GenPartialBallot()

for $C \in \mathcal{C}$ **do**

  $vv \xleftarrow{r} \mathbb{Z}_{n_v}$

  $Ctvv \leftarrow Ctvv \cup (C, vv)$

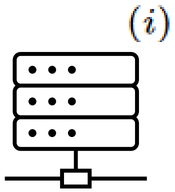
$ca \xleftarrow{r} \mathbb{Z}_{n_a}$

$p \xleftarrow{r} \mathbb{P}_{|\mathcal{C}|}$

return $(Ctvv, ca, p)$

randomized

(i)



ProVerif proof

Modeling

Challenges

XOR to combine secrets (e.g. Cred)

Permutations applied one-after-the-other

Algorithms executed only once per voter

Proof basis

Two servers, of which one is honest

Two voting options

Arbitrary number of elections and voters

XOR to combine secrets

Modeling XOR

ProVerif does not support XOR (commutative, associative)

In our protocol, Adversary knows all but one of the XORed values

→ model like a hash

```
(* XOR *)
type xor_part_t.
type xor_t.
fun xor_combine(xor_part_t, xor_part_t):xor_t.
```

```
let vote_code_1_verification = xor_combine(vote_code_1_verification_part1, vote_code_1_verification_part2) in
let vote_code_2_verification = xor_combine(vote_code_2_verification_part1, vote_code_2_verification_part2) in
insert Voter_VoteState(id, permuted_plain_vote_1, vote_code_1, vote_code_1_verification);
insert Voter_VoteState(id, permuted_plain_vote_2, vote_code_2, vote_code_2_verification);
```

Permutations applied one-after-the-other

Modeling permutations

Exhaustively list all possibilities

```
(* execute permutaton *)
fun permuted_plain_vote(plain_vote_t, nat): plain_vote_t
reduc forall x:bitstring;
    permuted_plain_vote(plain_vote_1, 0) = plain_vote_1
otherwise forall x:bitstring;
    permuted_plain_vote(plain_vote_1, 1) = plain_vote_2
otherwise forall x:bitstring;
    permuted_plain_vote(plain_vote_2, 0) = plain_vote_2
otherwise forall x:bitstring;
    permuted_plain_vote(plain_vote_2, 1) = plain_vote_1.
```

Algorithms executed only once per parameter

Modeling only once

Define restriction with Unique event

To use, apply unique() letfun with a corresponding label

```
(* to enforce unique execution of some parts (modelling choice; restriction). *)
type stamp_t.
type once_label_t.
event Unique(once_label_t, stamp_t).
restriction label: once_label_t, stamp,stamp': stamp_t;
    event(Unique(label, stamp)) && event(Unique(label, stamp')) ==> stamp = stamp'.
letfun unique(label: once_label_t) = new stamp: stamp_t; event Unique(label, stamp); ().
```

```
(* receive vote of voter (once) *)
in(adversary, (id:nat, vote_code:vote_code_t));
let () = unique(voting_CCV_once_per_id(id)) in
```

Privacy Proof

Proving equivalence

Voter(Alice, 0) | Voter(Bob, 1) ~ Voter(Alice, 1) | Voter(Bob, 0)

(Two honest voters vote differently depending on world)

```
VotingPhase_Voter(id_1, choice[plain_vote_1, plain_vote_2]) |
VotingPhase_Voter(id_2, choice[plain_vote_2, plain_vote_1]) |
```

```
in(id_driver_result(id_1), result_1: plain_vote_t);
in(id_driver_result(id_2), result_2: plain_vote_t);
out(c, (choice[result_1, result_2], choice[result_2, result_1]));
```

Proof terminates in 14s using ProVerif on a standard-issue laptop.

Verifiability Proof

Use Verifiability Framework (Cheval et al – CSF'23)

Implement 11 functions of the verifiability framework

Add restriction to prevent re-voting

```
expand Framework(
  Gen_Voter_Ident, Voting, Final_Check, Decrypt_Ballot, Gen_Voting_Data,
  gen_voter_data, is_valid, get_ident_from_ballot, is_valid_ballot, update_ballot, update_gbl,
  System
).

(* Honest voter should only attempt to vote once *)
restriction e_id:election_id, v_id:nat, nb_vote:nat, v:vote;
  event(WillVote(e_id,v_id,nb_vote,v)) ==> nb_vote = 1.
```

Proof terminates in 36s using ProVerif on a standard-issue laptop.

Verifiability Proof - Queries

```
(* Every counted vote from honest voter exactly as verified or voted *)
query e_id:election_id, v:vote, v_id:nat;
  event(Finish(e_id)) && inj-event(Counted(e_id,v)) ==>
      inj-event(HV(e_id,v_id)) && event(Verified(e_id,v_id,v))
    || inj-event(HNV(e_id,v_id)) && event(Voted(e_id,v_id,v))
    || inj-event(Corrupt(e_id,v_id))
.
```

```
(* Every verified vote counted *)
query e_id:election_id, v_id:nat, v:vote;
  event(Finish(e_id)) && inj-event(Verified(e_id,v_id,v)) ==>
    inj-event(Counted(e_id,v))
.
```

Summary

Protocol under proof

Short Voting Codes, Return Codes, Hashed Credentials

Deterministic trusted Setup, 1/n trusted Servers, no trust internet / device

Uses XOR, permutations, hashes, pk encryption, signatures, verifiable shuffle

Proven Property	Trust Assumptions	Proof basis	Proof
Vote Privacy Delaune et al – Journal CS '09	Setup, 1-out-of-n Servers	Custom (driver-based) n=2, 2 choices; unbounded voters/elections	✓
E2E-Verifiability Cortier et al – ESORICS'14	Setup, 1-out-of-n Servers	Verifiability framework (Cheval et al – CSF'23) n=2, 2 choices; up to 3 voters	✓

Sources

Sources
- Moser, Florian. "Short Voting Codes For Practical Code Voting." *arXiv preprint arXiv:2311.12710* (2023).
- Cheval, Vincent, Véronique Cortier, and Alexandre Debant. "Election Verifiability with ProVerif." *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. IEEE, 2023.
- Cortier, Véronique, et al. "Election verifiability for helios under weaker trust assumptions." *Computer Security-ESORICS 2014: 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II 19*. Springer International Publishing, 2014.
- Delaune, Stéphanie, Steve Kremer, and Mark Ryan. "Verifying privacy-type properties of electronic voting protocols." *Journal of Computer Security* 17.4 (2009): 435-487.