# Theorem Provers to Protect Democracy: Formally Verified Verifiable Vote Counting Software

Mukesh Tiwari,
University of Cambridge,
Cambridge

# Disclaimer

There is no perfect way —voting method— to combine preferences (Arrow's paradox).

| | |
|---|---|
| A | 1 |
| B | 2 |
| C | 3 |

Preferential Ballot

# Disclaimer

There are many ways to combine preferences
and Schulze is one way.

I am going to show a lot of Coq code

# Voting bug may have cost Rina Mercuri a seat on council

**Stephen Mudd**

Local News

A A A Comments

VOTING BUG: Rina Mercuri was almost certain to win a seat on council in 2012 but a software glitch saw it handed to Alison Balind instead. Picture: Anthony Stipo.


## LOCAL NEWS

1 Grants of up to $50,000 available for farmers hit by floods

2 MLHD ends year with 131 new COVID-19 cases, 'close contact' redefined

3 Six die as NSW records over 21,000 new cases in a day

4 PM announces major COVID-19 testing changes

# Claim

Formal methods can help us in protecting democracy

**Essential Meaning of *democracy***

**1** **:** a form of government in which people choose leaders by voting

// The nation has chosen *democracy* over monarchy.

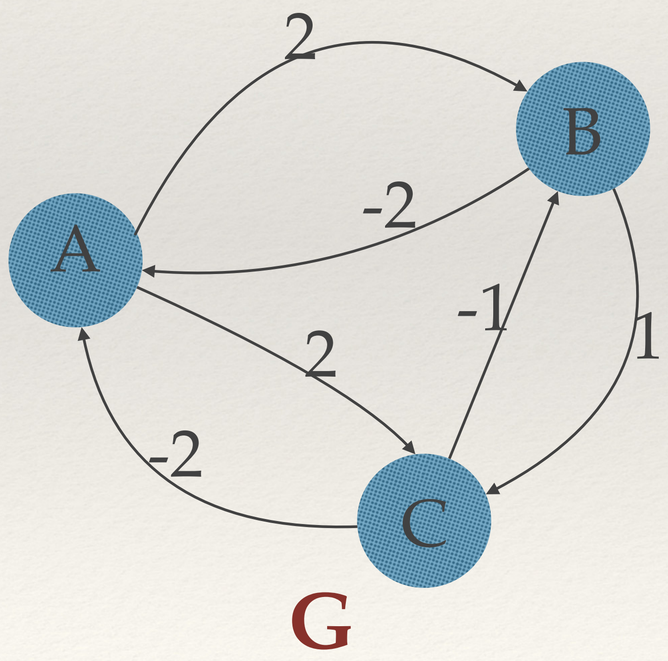
// the principles of *democracy*

# Schulze Method

| | C < B < A | C = B < A | C = B = A |
|---|---|---|---|
| A | 1 | 1 | 1 |
| B | 2 | 2 | 1 |
| C | 3 | 2 | 1 |
| | Ballot 1 | Ballot 2 | Ballot 3 |

Construct a margin function (matrix) $m : C \times C \to \mathbb{Z}$.

$$m(c,d) = \sharp\{b \in P \mid c >_b d\} - \sharp\{b \in P \mid d >_b c\}$$



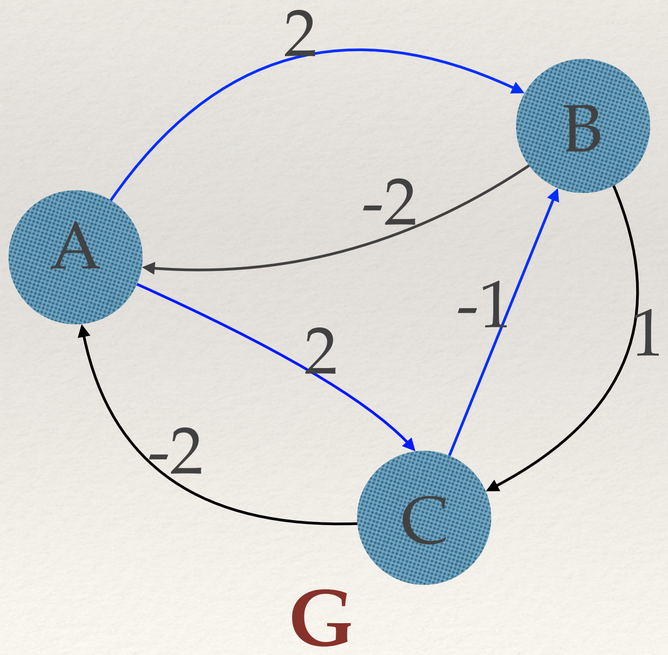

G

| | A | B | C |
|---|---|---|---|
| A | 0 | 2 | 2 |
| B | -2 | 0 | 1 |
| C | -2 | -1 | 0 |

Margin Matrix (m)

The strength, st, of a path in $G$ is

$$st(c_0, \ldots, c_{n+1}) = \min\{m(c_i, c_{i+1}) \mid 0 \leq i \leq n\}.$$




G

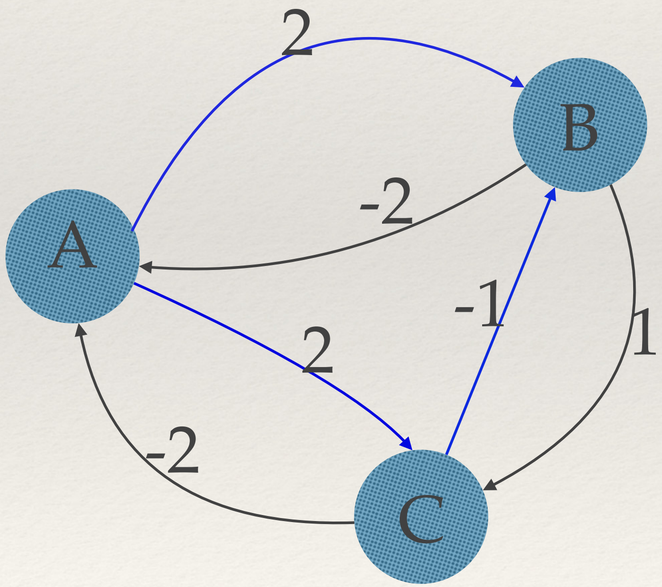
$$\text{st (A - B)} = \min \{\text{m (A, B)}\} = 2$$

$$\text{st (A - C - B)} = \min \{\text{m (A, C), m (C, B)}\} = \min \{2, -1\} = -1$$

- We compute the generalized margin, $M$, between two candidate $c$ $d$ as

$$M(c,d) = \max\{st(p) : \text{p is path from c to d in G}\}$$

$$M(A, B) = \max\{st(A, B), st(A\text{-}C\text{-}B)\}$$
$$= \max\{2, -1\}$$
$$= 2$$




G

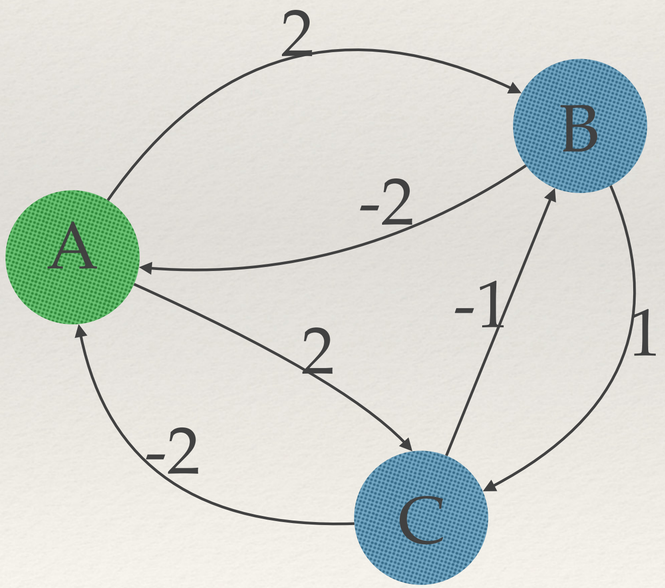
| | A | B | C |
|---|---|---|---|
| A | 0 | 2 | 2 |
| B | -2 | 0 | 1 |
| C | -2 | -1 | 0 |

Generalised Margin Matrix (M)

The winning set is defined as

$$W = \{c \in C : \forall d \in C \setminus \{c\}, M(c,d) \geq M(d,c)\}$$

A wins




G

| | A | B | C |
|---|---|---|---|
| A | 0 | 2 | 2 |
| B | -2 | 0 | 1 |
| C | -2 | -1 | 0 |

Generalised Margin Matrix (M)

# Vote-Counting-Software Requirements

Correctness

+

Verifiability

# Correctness (Specification)

```
(*
 Path m k c d means existence of a path p between two candidates c and d
 such that stength (p) >= k
 *)
Inductive Path (m : cand -> cand -> Z) (k : Z) : cand -> cand -> Prop :=
| unit c d : m c d >= k -> Path m k c d
| cons c e d : m c e >= k -> Path m k e d -> Path m k c d.
```

# Correctness (Specification)

```
(* Winning condition of Schulze Voting, notice that it is in Prop and
therefore no computational content. *)
Definition wins_prop (m : cand -> cand -> Z) (c: cand) : Prop :=
 forall d: cand, exists k: Z, Path m k c d /\
 (forall l, Path m l d c -> l <= k).
```

```
(* Dually, the notion of not winning: *)
 Definition loses_prop (m : cand -> cand -> Z) (c : cand) : Prop :=
  exists k: Z, exists  d: cand,
  Path m k d c /\ (forall l, Path m l c d -> l < k).
```

# Correctness (Specification)

```
(* Boolean function that determins if candidate c is winner or not *)
Definition schulze_winner (m : cand -> cand -> cand) (c : cand) : bool :=
(* definition omitted *)
```

```
(* Schulze winner function produces correct winner, according
 to wins_prop specification *)
Lemma correctness_proof_winner (m : cand -> cand) :
 forall c : cand, schulze_winner m c = true <-> wins_prop m c.
```

# Correctness (Specification)

```
(* Schulze winner function produces correct losers, according
 to loses_prop specification *)
Lemma correctness_proof_loser (m : cand -> cand) :
 forall c : cand, schulze_winner m c = false <-> loses_prop m c
```

# Verifiability (Data)

To demonstrate that a candidate **c** wins, for all competitors **d** we need to exhibit an integer **k**:

1. **Data** for the existence of a path from **c** to **d** with strength ≥ **k**

2. **Data** for the non-existence of a path from **d** to **c** that is stronger than **k**

# Path Evidence (Data) (Just for the Cambridge Audience)

$$V_k : \mathsf{Pow}(C \times C) \to \mathsf{Pow}(C \times C)$$
$$V_k(R) = \{(c,d) \in C^2 \mid m(c,d) \geq k \vee (m(c,e) \geq k \wedge (e,d) \in R, \text{ for some } e \in C)\}$$

$(c,d) \in LFP(V_k)$ implies there exists a path, that joins **c** and **d**, of strength $\geq$ k

# Path Evidence (Data) (Just for the Cambridge Audience)

Candidate $c$ is winner, if for every other candidate $d$ there is a $k$:

- $(c, d) \in LFP(V_k)$
- $(d, c) \notin LFP(V_{k+1})$

# Path Evidence (Data) (Just for the Cambridge Audience)

$$(c, d) \in LFP(V_k)$$

```
(*
 PathT m k c d means existence of a path p between two candidates c and d
 such that stength (p) >= k. It can be extracted as an OCaml code because
 it lives in Type universe
 *)
Inductive PathT (m : cand -> cand -> Z) (k : Z) : cand -> cand -> Type :=
| unitT c d : m c d >= k -> PathT m k c d
| consT c e d : m c e >= k -> PathT m k e d -> PathT m k c d.
```

```
Definition least_fixed_point (A : Type) (l : list A) (H : forall a : A, In a l)
           (V : Op A) (Hmon : mon V) :=  iter V (length l) empty_ss.
```

```
Lemma pathT_fixpoint : forall k n c d,
    Fixpoints.iter (V k) n Fixpoints.empty_ss (c, d) = true ->
    PathT k c d.
```

```
(1/2)
PathT k c d
```

# Path Evidence (Data) (Just for the Cambridge Audience)

$$(d, c) \notin LFP(V_{k+1})$$

$$(d, c) \notin LFP(V_{k+1}) \iff (d, c) \in C \times C \setminus LFP(V_{k+1})$$

$$(d, c) \in C \times C \setminus \mathrm{LFP}(V_{k+1}) \iff (d, c) \in \mathrm{GFP}(W_{k+1})$$

$$W_k(R) = \{(c, d) \in C^2 \mid m(c, d) < k \wedge (m(c, e) < k \vee (e, d) \in R, \text{ forall } e \in C)\}$$

# Verifiability(Data)

```
(*
a k-coclosed set is a set that is co-closed under W_k, or ignore co and read it:
a k-closed set is a set that is closed under W_k
*)
 Definition coclosed (m : cand -> cand -> Z) (k : Z) (p : (cand * cand) -> bool) :=
  forall x, p x = true -> W m k p x = true.
```

```
Definition W m (k : Z) (p : cand * cand -> bool) ((c d) : cand * cand) :=
  andb
   (m c d <? k)
   (forallb (fun e => orb (m (c, e) <? k) (p (e, d))) cand_all)
```

# Verifiability(Data)

```
(* type-level winning condition in Schulze counting *)
Definition wins_type m c :=
forall d : cand, existsT (k : Z),
 ((PathT m k c d) *
 (existsT (f : (cand * cand) -> bool), f (d, c) = true /\ coclosed m (k + 1) f))
```

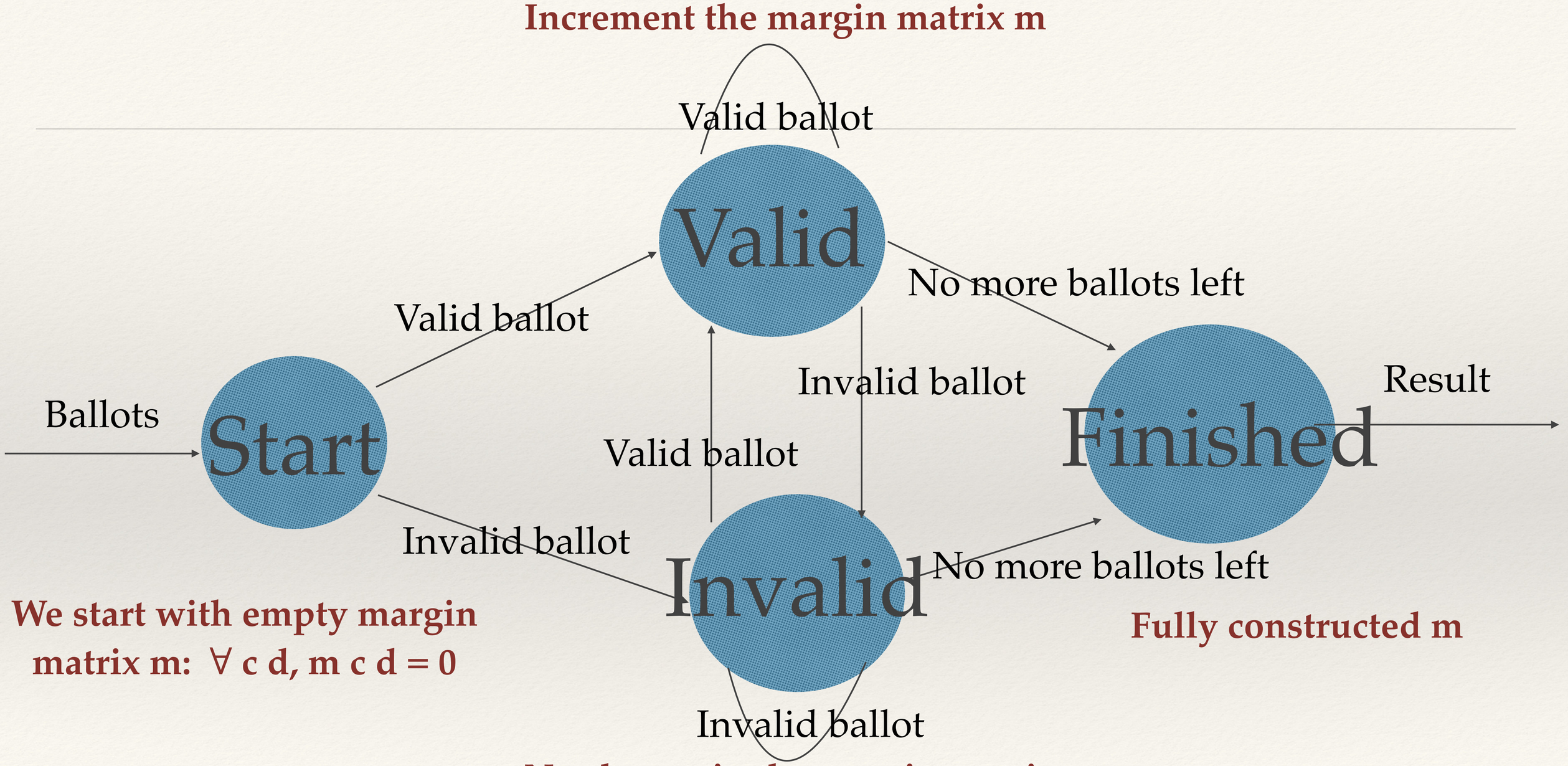
```
(1/1)
existsT f : cand * cand -> bool,
  f (d, c) = true /\ coclosed (k + 1) f
```

```
(* second one *)
exists (Fixpoints.greatest_fixed_point (cand * cand) (all_pairs cand_all)
                                       (all_pairs_universal cand_all cand_fin)
                                       (W (k + 1)) (monotone_operator_w (k + 1))).
```

# Confession

```
(* the type level winning condition can be reconstruced from *)
(* propositional knowledge of winning *)
 Lemma wins_prop_type (m : cand -> cand -> Z) :
  forall c, wins_prop m c -> wins_type m c.
```

```
(* dually, the type-level information witnessing winners *)
(* entails prop-level knowledge. *)
 Lemma wins_type_prop (m : cand -> cand -> Z) :
  forall c, wins_type m c -> wins_prop m c.
```


Increment the margin matrix m
Valid ballot
Valid
No more ballots left
Valid ballot
Invalid ballot
Result
Ballots
Start
Finished
Valid ballot
Invalid ballot
Invalid
No more ballots left
We start with empty margin matrix m: ∀ c d, m c d = 0
Fully constructed m
Invalid ballot
No change in the margin matrix m

# Schulze Voting as Inductive Type

```
Inductive Count (bs : list ballot) : State -> Type :=
| Bootstrap_counting us m : us = bs -> (forall c d, m c d = 0) ->
                  Count bs (partial (us, []) m) (* zero margin *)
```

# Schulze Voting as Inductive Type

```
| Valid_ballot u us m nm inbs :
   Count bs (partial (u :: us, inbs) m) ->
  (forall c, (u c > 0)%nat) -> (* u is valid *)
       (forall c d : cand,
        ((u c < u d)%nat -> nm c d = m c d + 1) (* c preferred to d *) /\
        ((u c = u d)%nat -> nm c d = m c d)     (* c, d rank equal  *) /\
        ((u c > u d)%nat -> nm c d = m c d - 1))(* d preferred to c *) ->
        Count bs (partial (us, inbs) nm)
```

# Schulze Voting as Inductive Type

```
| Invalid_ballot u us m inbs :
  Count bs (partial (u :: us, inbs) m) ->
  (exists c, (u c = 0)%nat) (* u is invalid *) ->
  Count bs (partial (us, u :: inbs) m)
```

# Schulze Voting as Inductive Type

```
| Finished (m : cand -> cand -> Z) inbs
   (w : cand -> bool)
   (d : (forall c, (wins_type m c) + (loses_type m c))):
   Count bs (partial ([], inbs) m) (* no ballots left  *) ->
   (forall c, w c = true <-> (exists x, d c = inl x)) ->
   (forall c, w c = false <-> (exists x, d c = inr x)) ->
   Count bs (winners w).
```

```
(* The main theorem: for every list of ballots, we can find a boolean function that decides
 winners, together with evidences of the correctness of this determination *)
Definition schulze_winners (bs : list ballot) :
  existsT (f : cand -> bool) (p : Count bs (winners f)), True :=
  let (i, t) := all_ballots_counted bs in
  let (m, p) := t in
  existT _ (c_wins m) (existT _ (fin _ _ _ _ (wins_loses_type_dec m) p
                                  (c_wins_true_type m) (c_wins_false_type m)) I).
```

# Demo

# The Ballot Identification Problem

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 1 | 4 | 6 | 7 | 5 | 11 | 9 | 10 | 15 | 14 | 8 | 13 | 12 |

# Solution

1. Hide the preferences by encrypting ballots

2. Prove that vote counting method on encrypted ballots is correct

3. Produce Zero-Knowledge-Proof to ensure verifiability

# Encrypted Ballot

$$C < B < A$$

| | A | B | C |
|---|---|---|---|
| A | E(0, r1) | E(1,r2) | E(1, r3) |
| B | E(-1, r4) | E(0, r5) | E(1, r6) |
| C | E(-1, r7) | E(-1, r8) | E(0, r9) |

$$E(v, r) = (g^r, g^v \cdot h^r)$$

| | A | B | C |
|---|---|---|---|
| A | E(0, r1) | E(1,r2) | E(1, r3) |
| B | E(-1, r4) | E(0, r5) | E(1, r6) |
| C | E(-1, r7) | E(-1, r8) | E(0, r9) |

Ballot 1: C < B < A

| | A | B | C |
|---|---|---|---|
| A | E(0, r11) | E(-1, r12) | E(-1, r13) |
| B | E(1, r14) | E(0, r15) | E(-1, r16) |
| C | E(1, r17) | E(1, r18) | E(0, r19) |

Ballot 2: A < B < C

$$Enc(v_1, r_1) = (g^{r_1}, g^{v_1} \cdot h^{r_1})$$
$$Enc(v_2, r_2) = (g^{r_2}, g^{v_2} \cdot h^{r_2})$$
$$Enc(v_1, r_1) \cdot Enc(v_2, r_2) = (g^{r_1+r_2}, g^{v_1+v_2} \cdot h^{r_1+r_2})$$

# Challenge

What is the ordering of candidates in this ballot?

| | A | B | C |
|---|---|---|---|
| A | E(0, r1) | E(**1**, r2) | E(**1**, r3) |
| B | E(**1**, r4) | E(0, r5) | E(-1, r6) |
| C | E(**1**, r7) | E(-1, r8) | E(0, r9) |

# Challenge

## What about this one?

| | A | B | C |
|---|---|---|---|
| A | E(0, r1) | E(100, r2) | E(100, r3) |
| B | E(-100, r4) | E(0, r5) | E(1, r6) |
| C | E(-100, r7) | E(-1, r8) | E(0, r9) |

# Challenge

Deciding if a ballot is valid or not

For **every ballot**, the counting authority generates a **secret permutation**, but it's **publicly verifiable** that it's a valid permutation (Zero-Knowledge-Proof)

[A -> C, C -> B, B -> A]

| | A | B | C |
|---|---|---|---|
| A | E(0, r1) | E(100, r2) | E(100, r3) |
| B | E(-1, r4) | E(0, r5) | E(1, r6) |
| C | E(-1, r7) | E(-1, r8) | E(0, r9) |

Original Ballot

| | A | B | C |
|---|---|---|---|
| A | **E(1, r10)** * E(100, r2) | **E(1, r11)** * E(100, r3) | **E(1, r12)** * E(0, r1) |
| B | | | |
| C | | | |

Row Shuffled Ballot

+ Zero Knowledge Proof

[A -> C, C -> B, B -> A]

| | A | B | C |
|---|---|---|---|
| A | E(1, r10) * E(100, r2) | E(1, r11) * E(100, r3) | E(1, r12) * E (0, r1) |
| B | | | |
| C | | | |

Row Shuffled Ballot

| | A | B | C |
|---|---|---|---|
| A | | | |
| B | | | |
| C | | | |

Row-Column Shuffled Ballot

Zero Knowledge Proof

# Decryption with Zero-Knowledge-Proof

| | A | B | C |
|---|---|---|---|
| A | | | |
| B | | | |
| C | | | |

Row-Column Shuffled Ballot

| | A | B | C |
|---|---|---|---|
| A | 100 | 0 | 100 |
| B | 1 | 1 | 1 |
| C | -1 | -1 | 0 |

Decrypted Row-Column Shuffled Ballot

Zero Knowledge Proof

# Schulze Voting as Inductive Type

```
(* Inductive type for counting. Indexed over a given Group, and list of ballots
   to be counted *)
Inductive ECount (grp : Group) (bs : list eballot) : EState -> Type :=
| ecax (us : list eballot) (encm : cand -> cand -> ciphertext)
       (decm : cand -> cand -> plaintext)
       (zkpdec : cand -> cand -> DecZkp) :
    us = bs ->
    (forall c d : cand, decm c d = 0) ->
    (forall c d, verify_zero_knowledge_decryption_proof
              grp (decm c d) (encm c d) (zkpdec c d) = true) ->
    ECount grp bs (epartial (us, []) encm)
```

# Schulze Voting as Inductive Type

```
| ecvalid (u : eballot) (v : eballot) (w : eballot)
         (b : pballot) (zkppermuv : cand -> ShuffleZkp)
         (zkppermvw : cand -> ShuffleZkp) (zkpdecw : cand -> cand -> DecZkp)
         (cpi : Commitment) (zkpcpi : PermZkp)
         (us : list eballot) (m nm : cand -> cand -> ciphertext)
         (inbs : list eballot) :
    ECount grp bs (epartial (u :: us, inbs) m) ->
    matrix_ballot_valid b ->
    (* commitment proof *)
    verify_permutation_commitment grp (List.length cand_all) cpi zkpcpi = true ->
    (forall c, verify_row_permutation_ballot grp u v cpi zkppermuv c = true)  ->
    (forall c, verify_col_permutation_ballot grp v w cpi zkppermvw c = true)  ->
    (forall c d, verify_zero_knowledge_decryption_proof
              grp (b c d) (w c d) (zkpdecw c d) = true) (* b is honest decryption of w *) ->
    (forall c d, nm c d = homomorphic_addition grp (u c d) (m c d)) ->
    ECount grp bs (epartial (us, inbs) nm)
```

# Schulze Voting as Inductive Type

```
| ecinvalid (u : eballot) (v : eballot) (w : eballot)
          (b : pballot) (zkppermuv : cand -> ShuffleZkp)
          (zkppermvw : cand -> ShuffleZkp) (zkpdecw : cand -> cand -> DecZkp)
          (cpi : Commitment) (zkpcpi : PermZkp)
          (us : list eballot) (m : cand -> cand -> ciphertext)
          (inbs : list eballot) :
    ECount grp bs (epartial (u :: us, inbs) m) ->
    ~matrix_ballot_valid b ->
    (* commitment proof *)
    verify_permutation_commitment grp (List.length cand_all) cpi zkpcpi = true  ->
    (forall c, verify_row_permutation_ballot grp u v cpi zkppermuv c = true) ->
    (forall c, verify_col_permutation_ballot grp v w cpi zkppermvw c = true) ->
    (forall c d, verify_zero_knowledge_decryption_proof
              grp (b c d) (w c d) (zkpdecw c d) = true) (* b is honest decryption of w *) ->
    ECount grp bs (epartial (us, (u :: inbs)) m)
```

# Schulze Voting as Inductive Type

```
| ecdecrypt inbs (encm : cand -> cand -> ciphertext)
              (decm : cand -> cand -> plaintext)
              (zkp : cand -> cand -> DecZkp) :
    ECount grp bs (epartial ([], inbs) encm) ->
    (forall c d, verify_zero_knowledge_decryption_proof
                 grp (decm c d) (encm c d) (zkp c d) = true) ->
    ECount grp bs (edecrypt decm)
| ecfin dm w (d : (forall c, (wins_type dm c) + (loses_type dm c))) :
    ECount grp bs (edecrypt dm) ->
    (forall c, w c = true <-> (exists x, d c = inl x)) ->
    (forall c, w c = false <-> (exists x, d c = inr x)) ->
    ECount grp bs (ewinners w).
```

```
(* The main theorem: for every list of ballots, we can find a boolean function that decides
 winners, together with evidences of the correctness of this determination *)
Lemma pschulze_winners (bs : list eballot) :
  existsT (f : cand -> bool), ECount grp bs (ewinners f).
Proof.
  destruct (decrypt_margin bs) as [dm Hecount].
  exists (c_wins dm).
  pose proof (ecfin grp bs dm (c_wins dm) (wins_loses_type_dec dm) Hecount).
  pose proof (X (c_wins_true_type dm) (c_wins_false_type dm)).
  auto.
Defined.
```

# Collaboration Opportunities

1. Formalising all the cryptographic primitives
2. Developing a formally verified verifier
3. Verifying social choice properties
4. Anonymous voting system based on Ring Signatures
5. Anonymous file dropping system to protect whistle blowers

# Related Work

Pattinson, D. and Tiwari, M., 2017. Schulze Voting as Evidence carrying computation. In Proc. ITP 2017, vol. 10499 of Lecture Notes in Computer Science, 410–426. Springer.

Milad K. Ghale, Rajeev Goré, Dirk Pattinson, Mukesh Tiwari. Modular Formalisation and Verification of STV Algorithms. E-Vote-ID 2018: 51-66

# Related Work

Lyria Bennett Moses, Rajeev Goré, Ron Levy, Dirk Pattinson, Mukesh Tiwari. No More Excuses: Automated Synthesis of Practical and Verifiable Vote-Counting Programs for Complex Voting Schemes. E-VOTE-ID 2017: 66-83

Thomas Haines, Dirk Pattinson, and Mukesh Tiwari. 2019. Verifiable Homomorphic Tallying for the Schulze Vote Counting Scheme. 11th International Conference Verified Software: Theories, Tools, and Experiments. VSTTE 2019

# Related Work

Thomas Haines, Rajeev Goré, and Mukesh Tiwari. 2019. Verified Verifiers for Verifying Elections. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)

# Thank You