GOING FURTHER WITH PROVERIF

VeriCrypt 2021

Vincent Cheval
Inria

18/12/2021

Outline

A. Additional modeling techniques
   1. Equational Theory vs Rewrite rules
   2. Equivalence properties
   3. Memory cell and locks for stateful protocols
B. Dealing with « cannot be proved »
   1. Adding precision
   2. Trace with assumption
   3. Proof by induction
   4. Lemma
C. Dealing with non-termination

Equational theory vs Rewrite rules

Strengths and weaknesses of rewrite rules

+ Verification efficient
+ Very expressive with otherwise

```
fun ifthenelse(bool,bitstring,bitstring):bitstring
reduc
  forall x,y:bitstring; ifthenelse(true,x,y) = x
  otherwise forall b:bool,x,y:bitstring; ifthenelse(b,x,y) = y.
```

the term

ifthenelse(true,m,decrypt(a,k))

fails

```
fun lazy_ite(bool,bitstring,bitstring):bitstring
reduc
  forall x:bitstring; y:bitstring or fail; lazy_ite(true,x,y) = x
  otherwise forall b:bool,x:bitstring or fail,y:bitstring; lazy_ite(b,x,y) = y.
```

Equational theory vs Rewrite rules

Strengths and weaknesses of rewrite rules

+ Verification efficient
+ Very expressive with otherwise

- Cannot call itself

Algrebraic properties that cannot be modeled with rewrite rules in ProVerif

$$\mathsf{dec}(\mathsf{enc}(x, y), y) = x \quad \text{with} \quad \mathsf{enc}(\mathsf{dec}(x, y), y) = x$$

$$\mathsf{exp}(\mathsf{exp}(\mathsf{g}, x), y) = \mathsf{exp}(\mathsf{exp}(\mathsf{g}, y), x)$$

Diffie-Hellman

Equational theory vs Rewrite rules

Strengths and weaknesses of equational theory

+ Extremely expressive

- Makes the verification slow
- Not all equational theory can be handled (may not terminate from the start)

```
fun enc(G, passwd): G.
fun dec(G, passwd): G.
equation forall x: G, y: passwd; dec(enc(x,y),y) = x.
equation forall x: G, y: passwd; enc(dec(x,y),y) = x.
```

```
const g: G.
fun exp(G, exponent): G.
equation forall x: exponent, y: exponent; exp(exp(g, x), y) = exp(exp(g, y), x).
```

Equivalence properties

Type of security properties

Reachability

Bad event in one system

Authentication

Secrecy

Equivalence properties

Type of security properties

Reachability

Bad event in one system

- Authentication
- Secrecy

Equivalence

Privacy as indistinguishability

- Anonymity
- Vote privacy
- Unlinkability

Equivalence properties

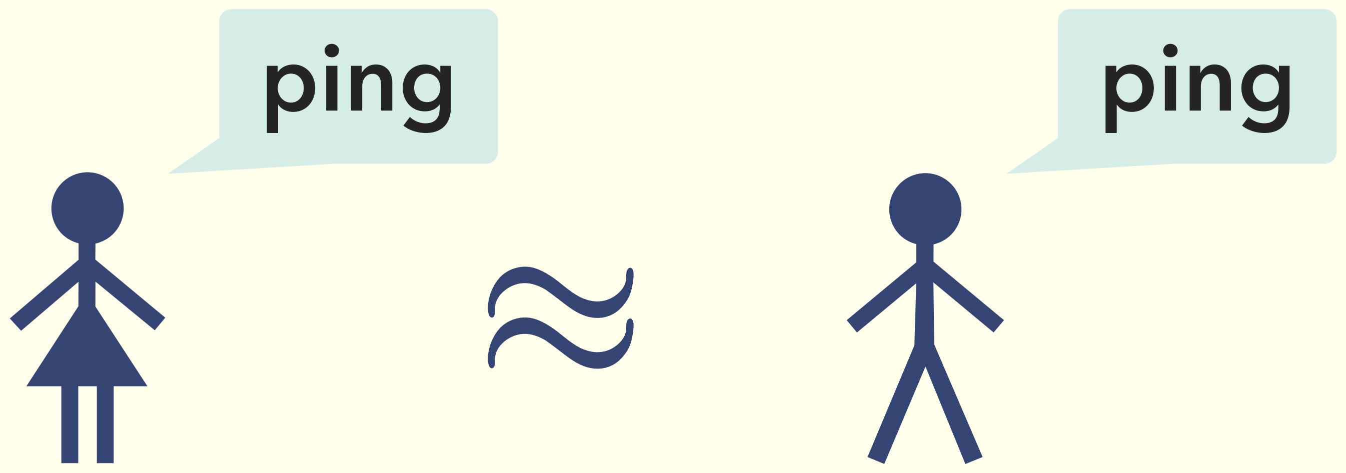
Indistinguishability

of two situations where the private attribute differs

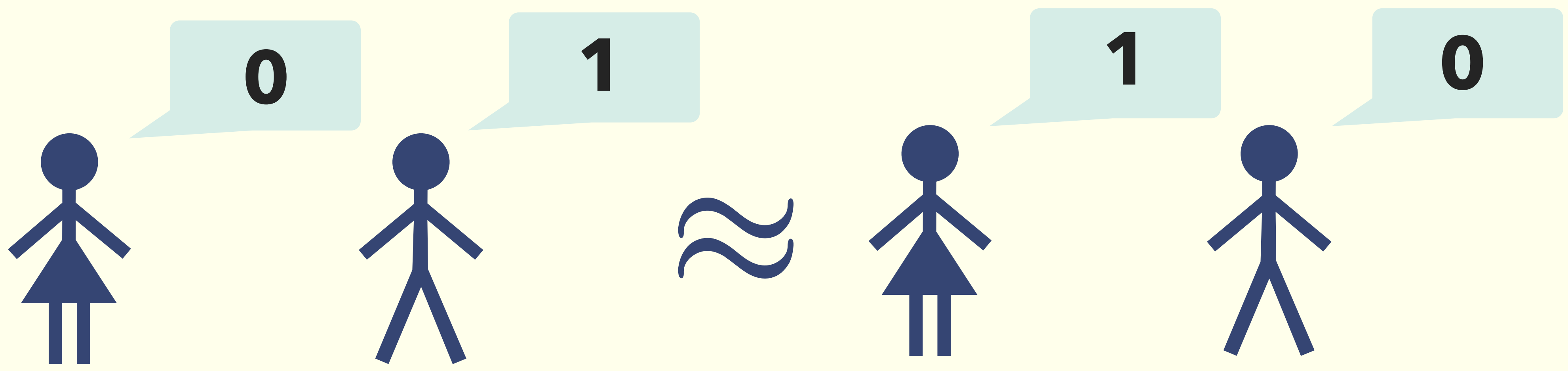
Equivalence properties

Indistinguishability

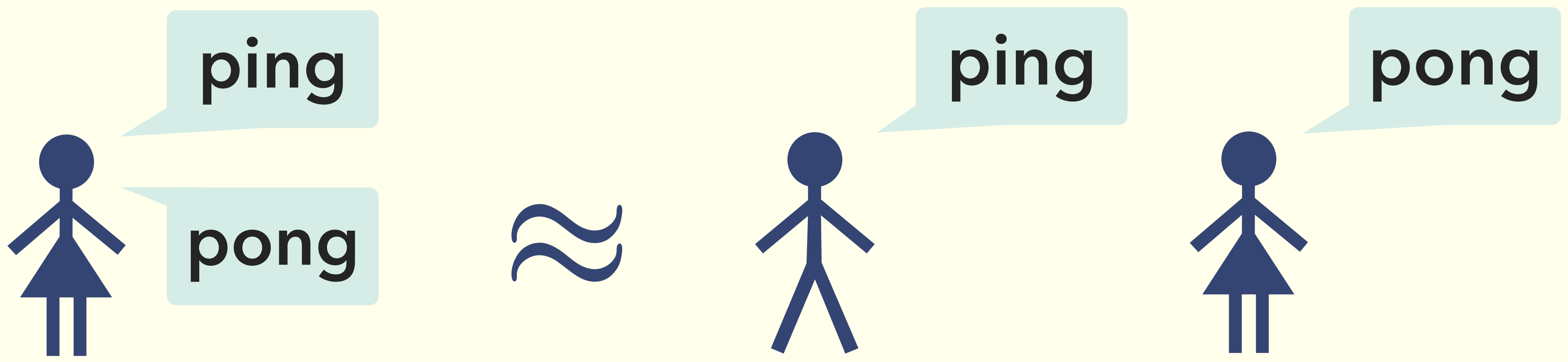
of two situations where the private attribute differs



Anonymity

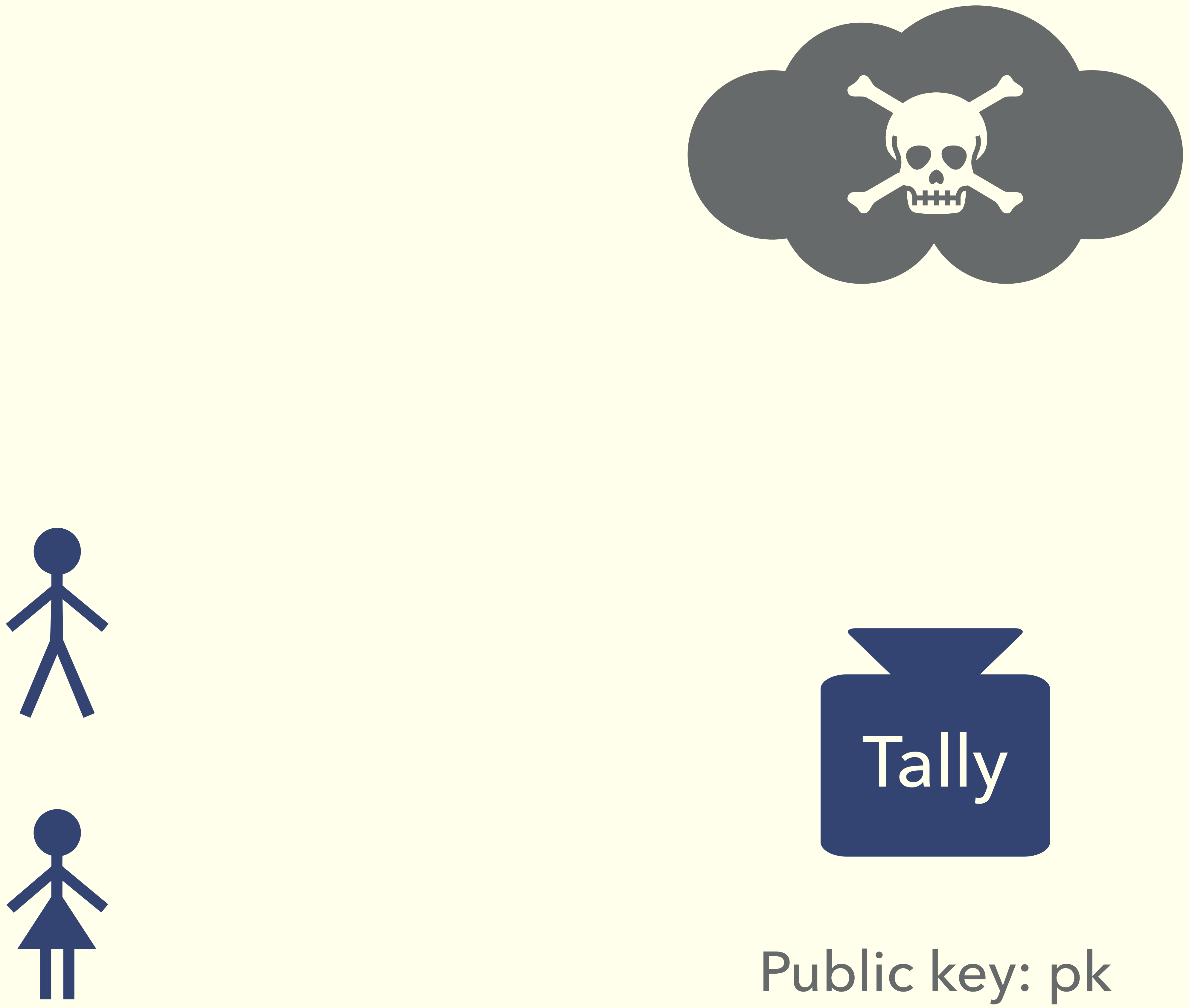


Vote privacy

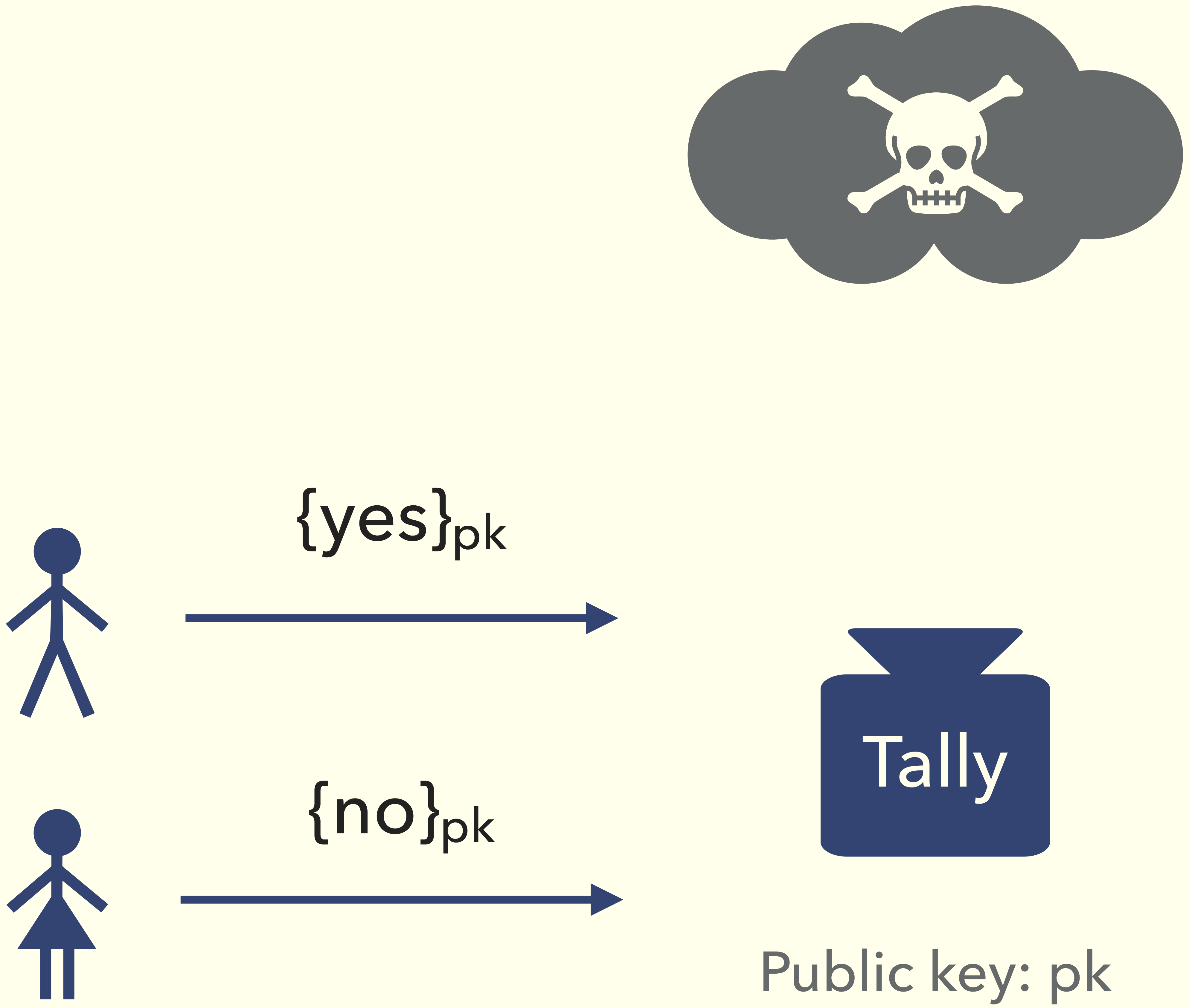


Unlinkability

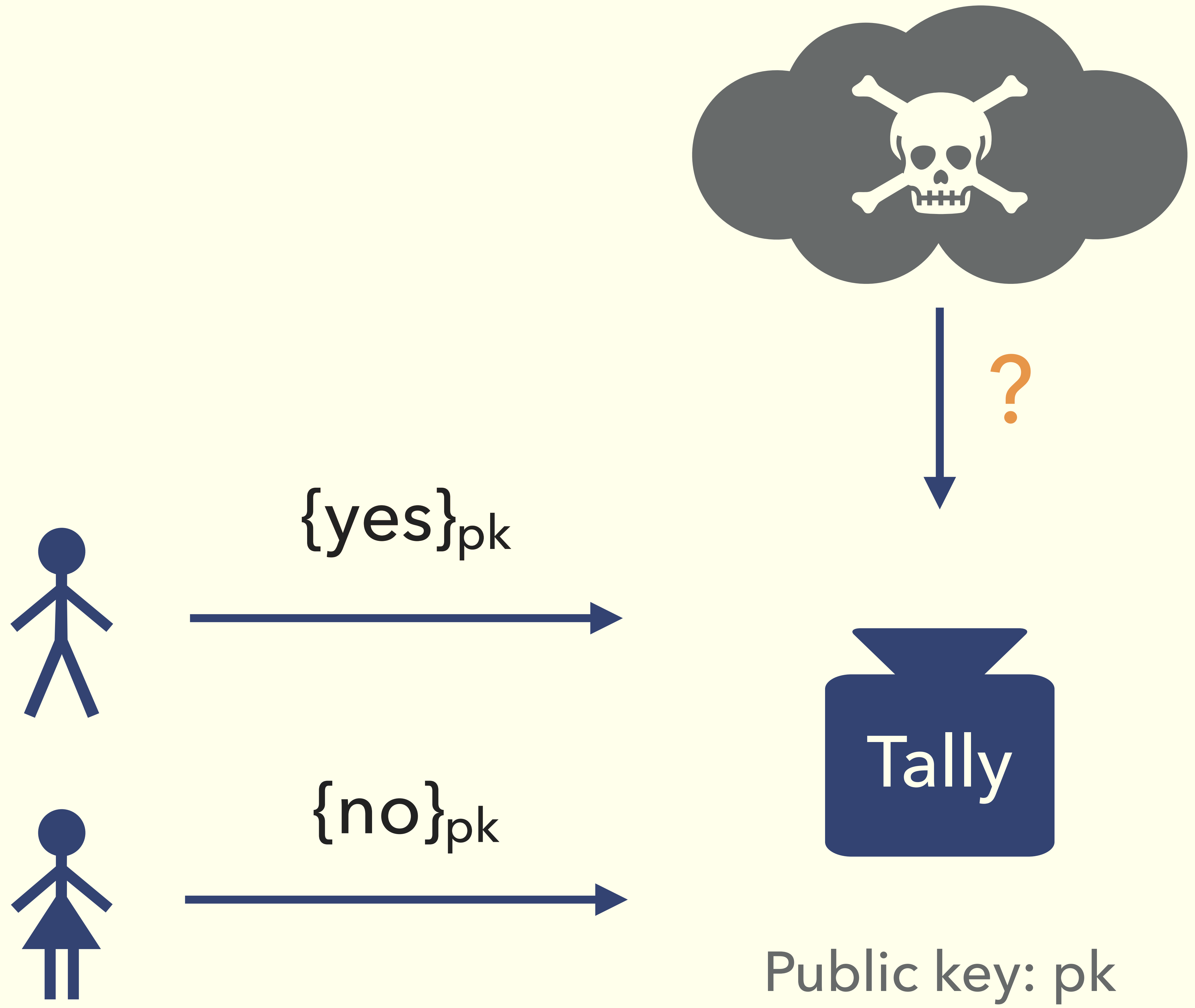
A simple e-voting protocol



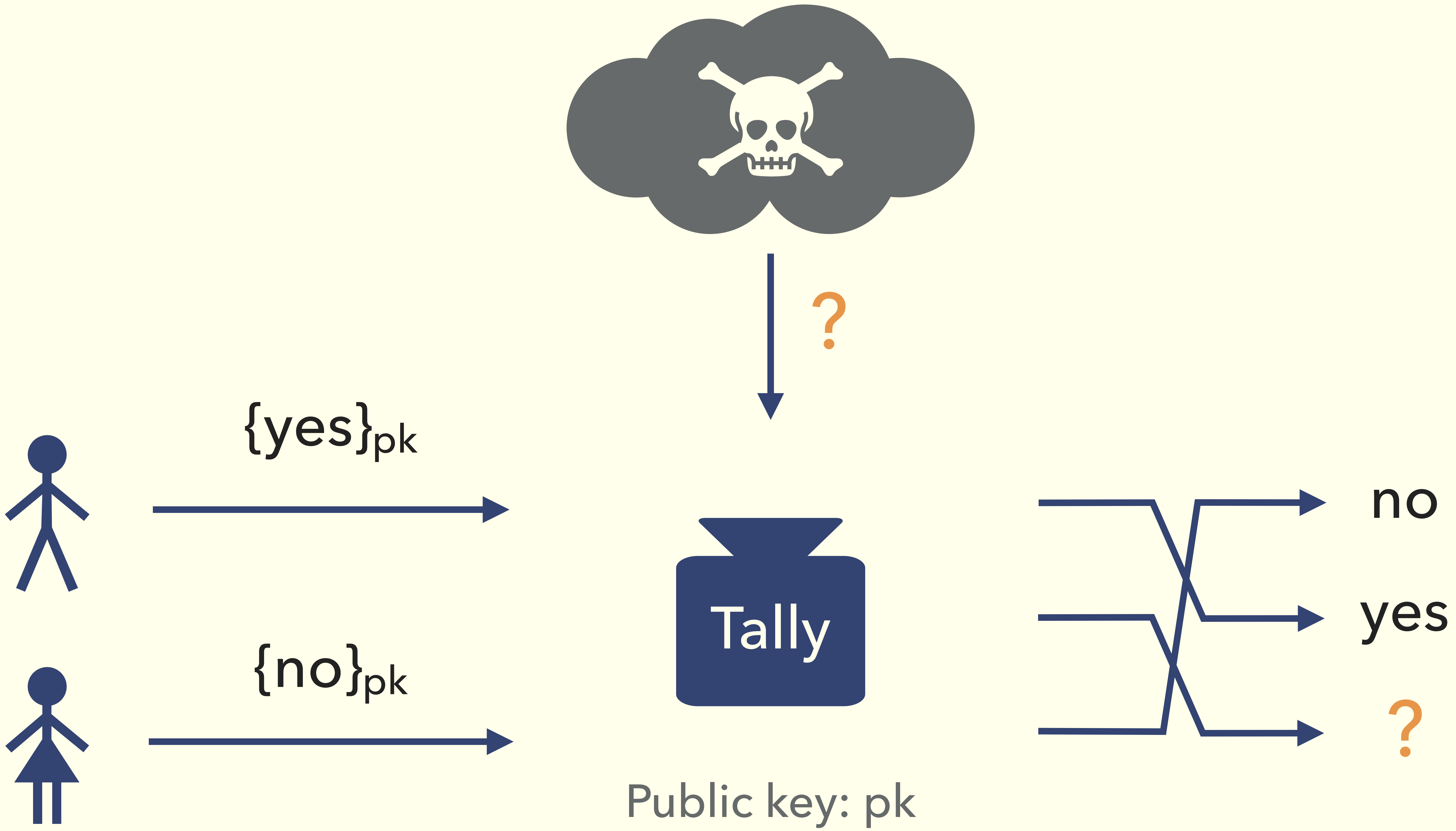
A simple e-voting protocol



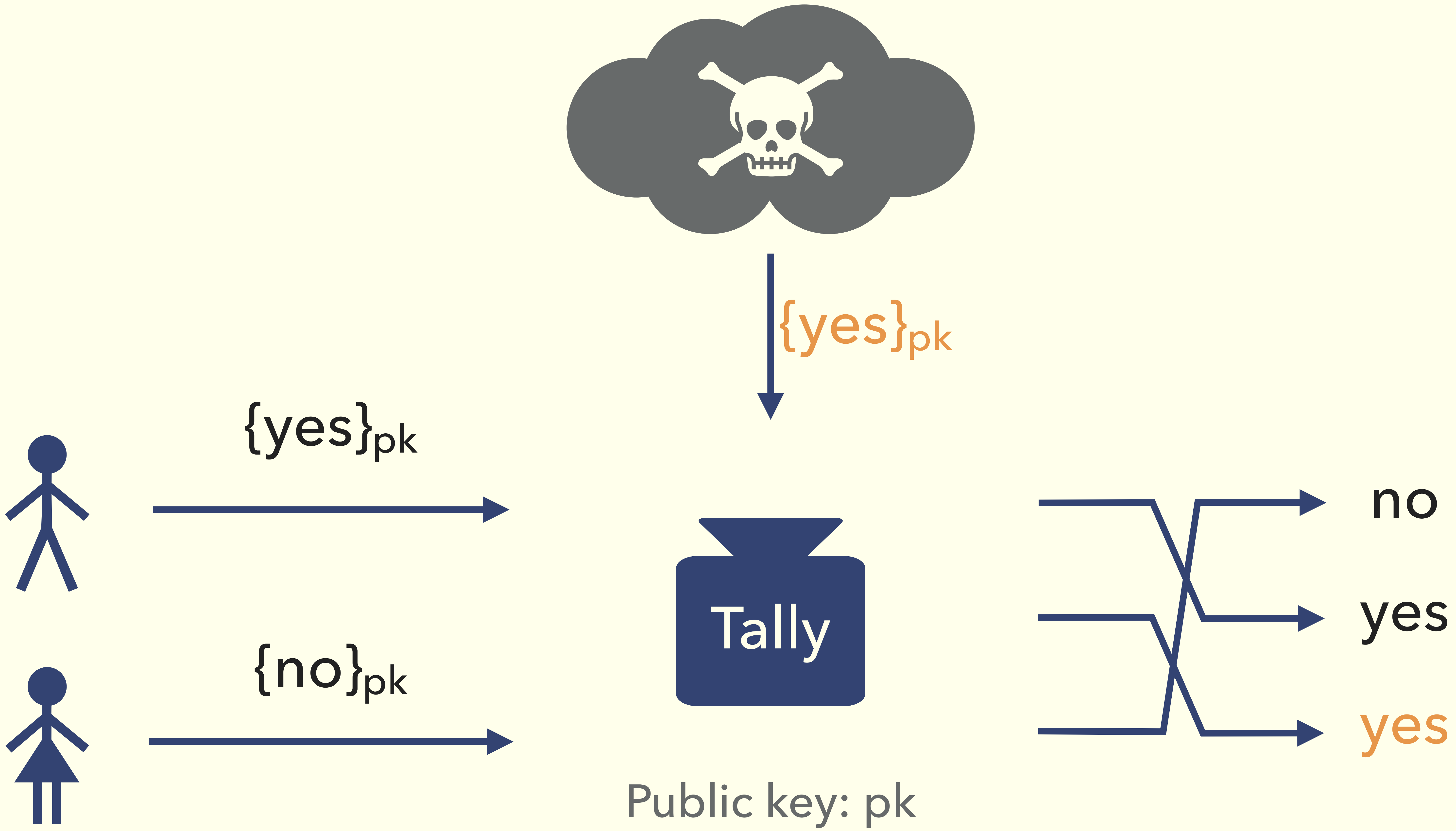
A simple e-voting protocol



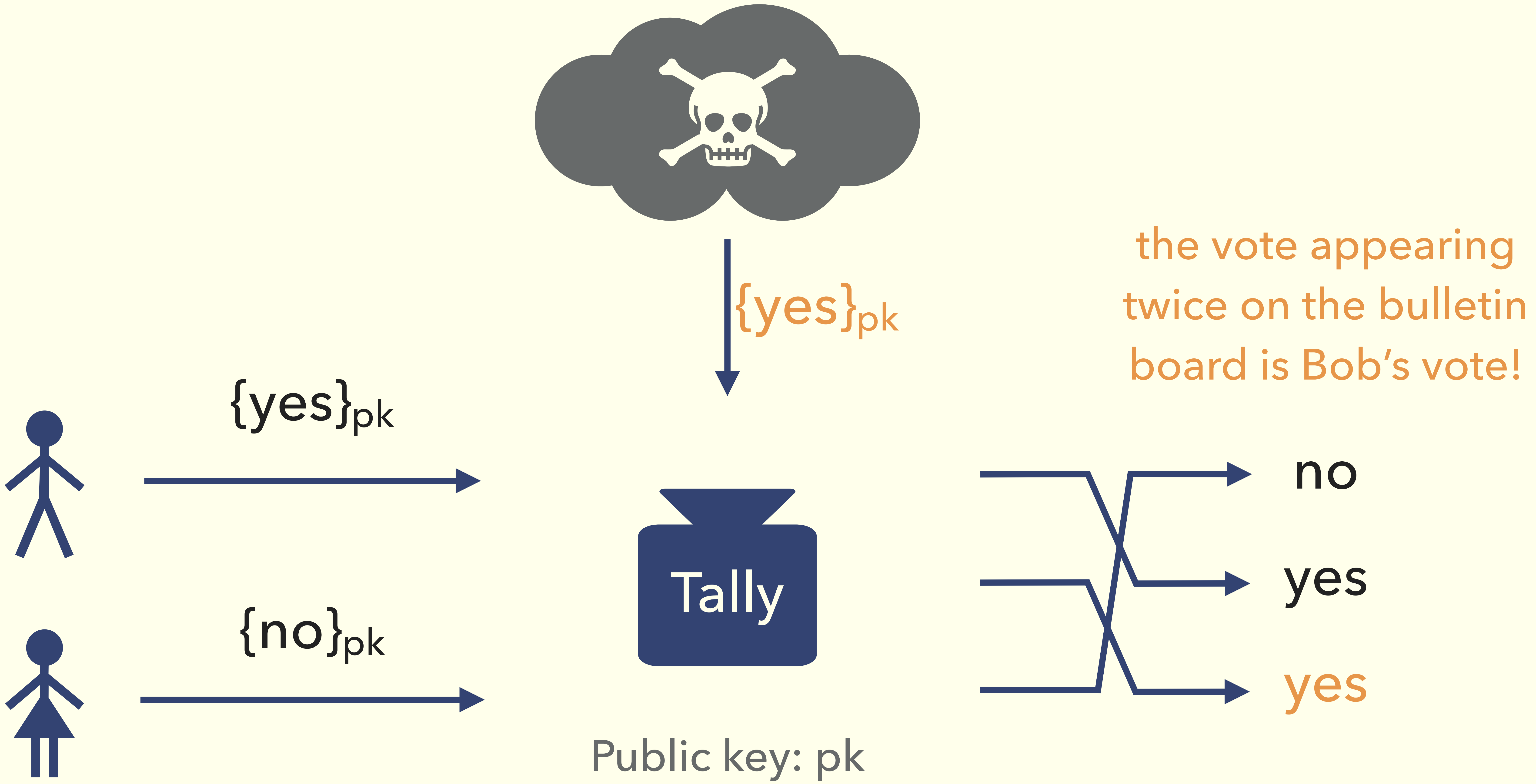
A simple e-voting protocol



A simple e-voting protocol

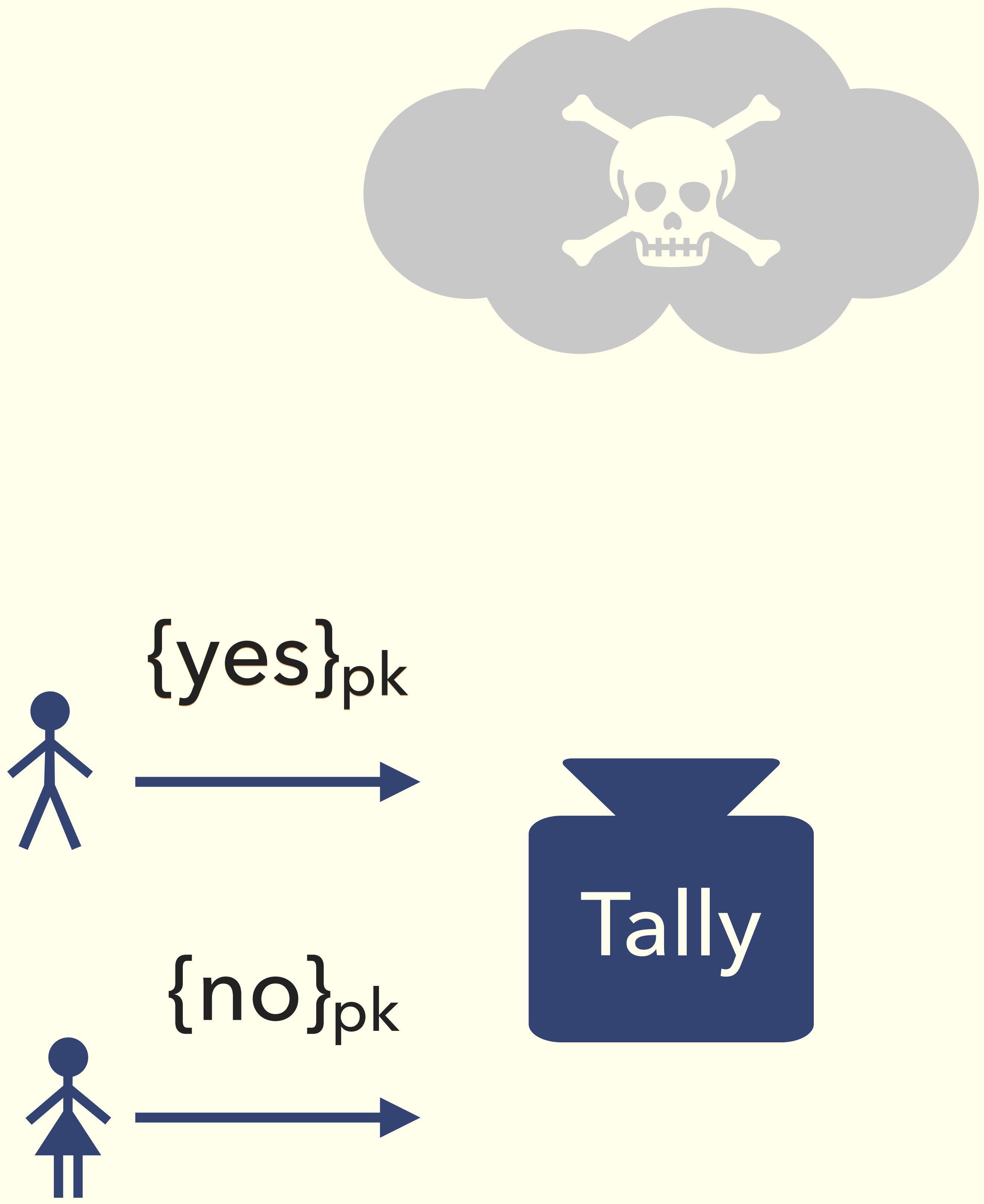


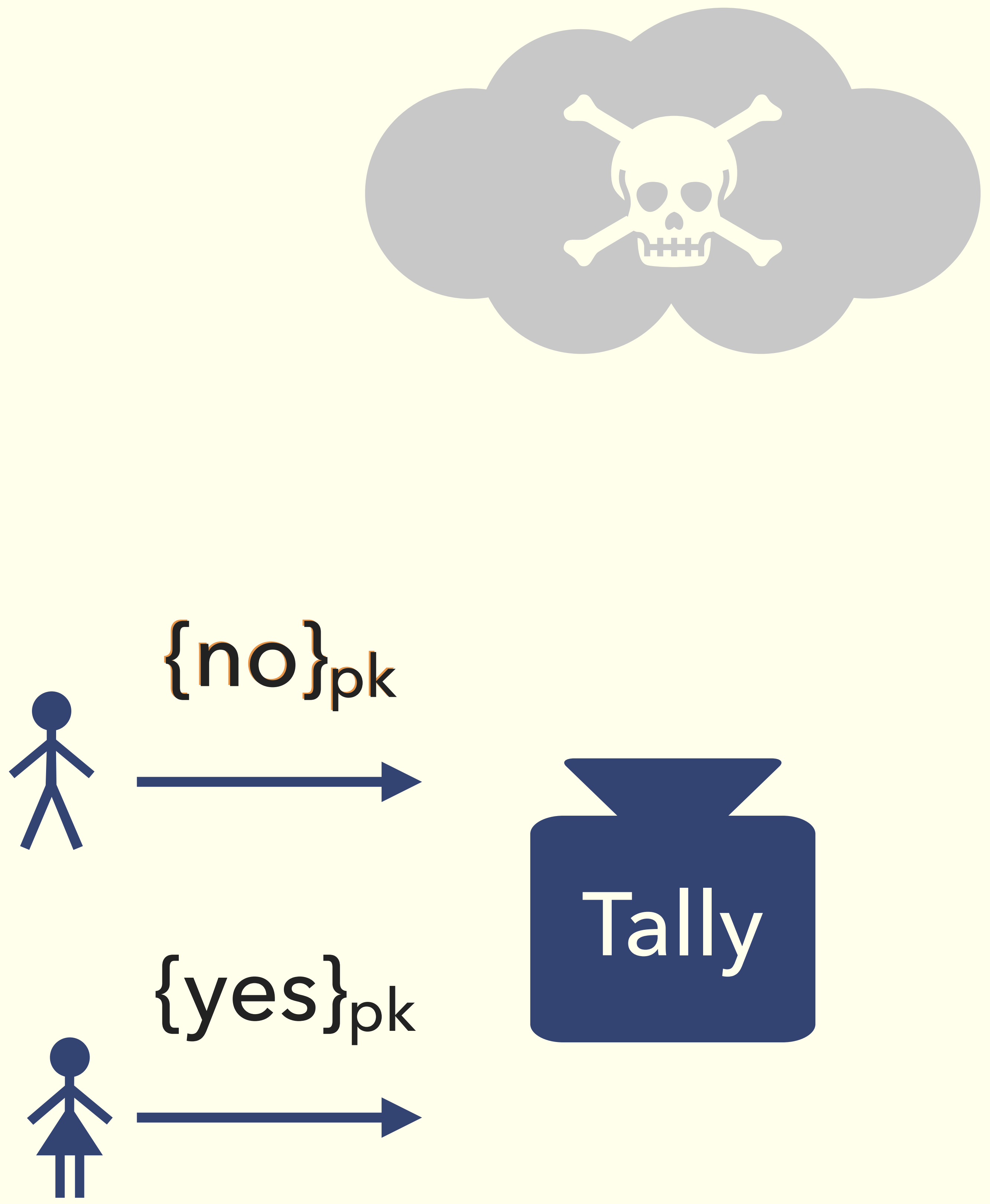
A simple e-voting protocol



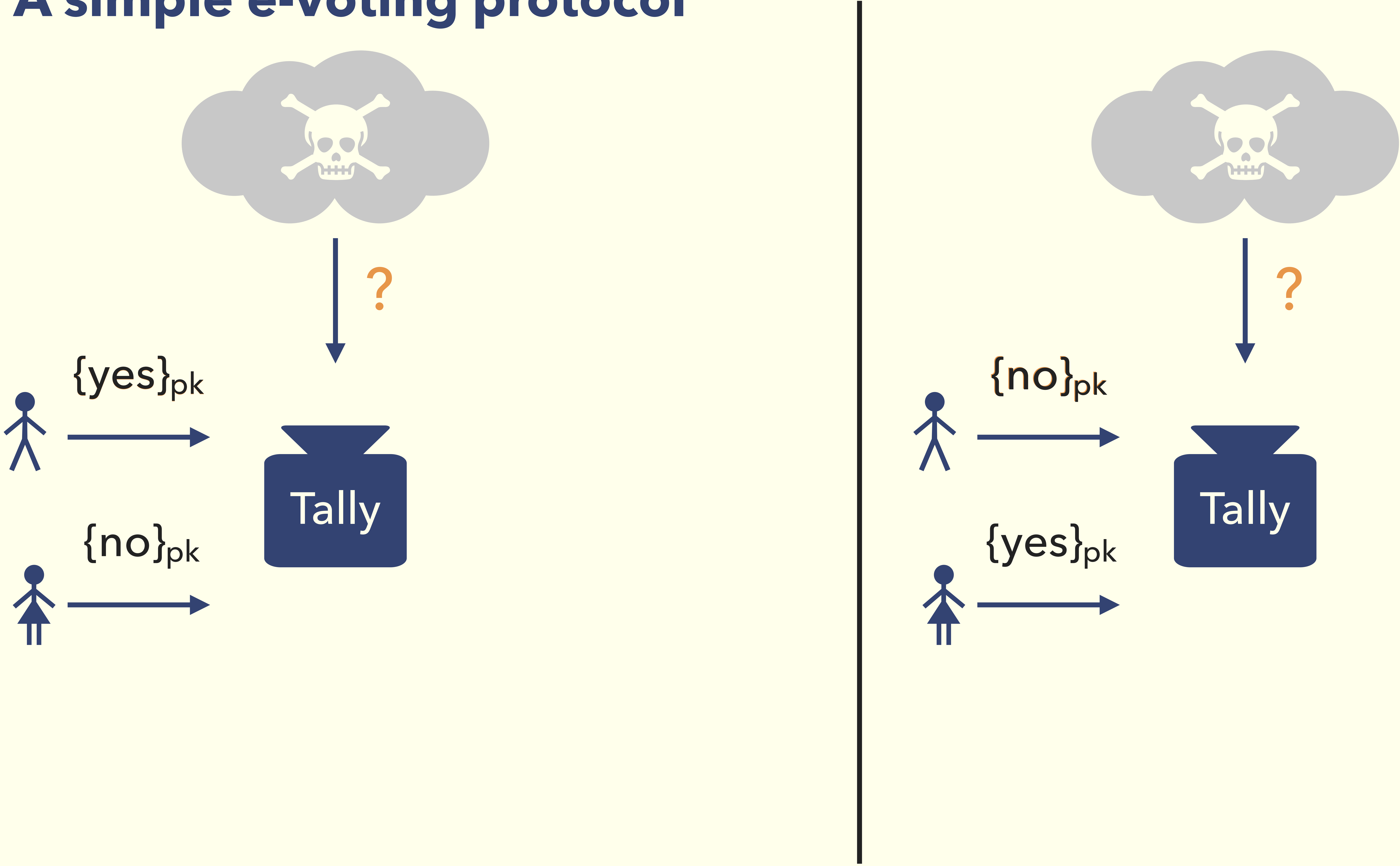
A simple e-voting protocol

A simple e-voting protocol

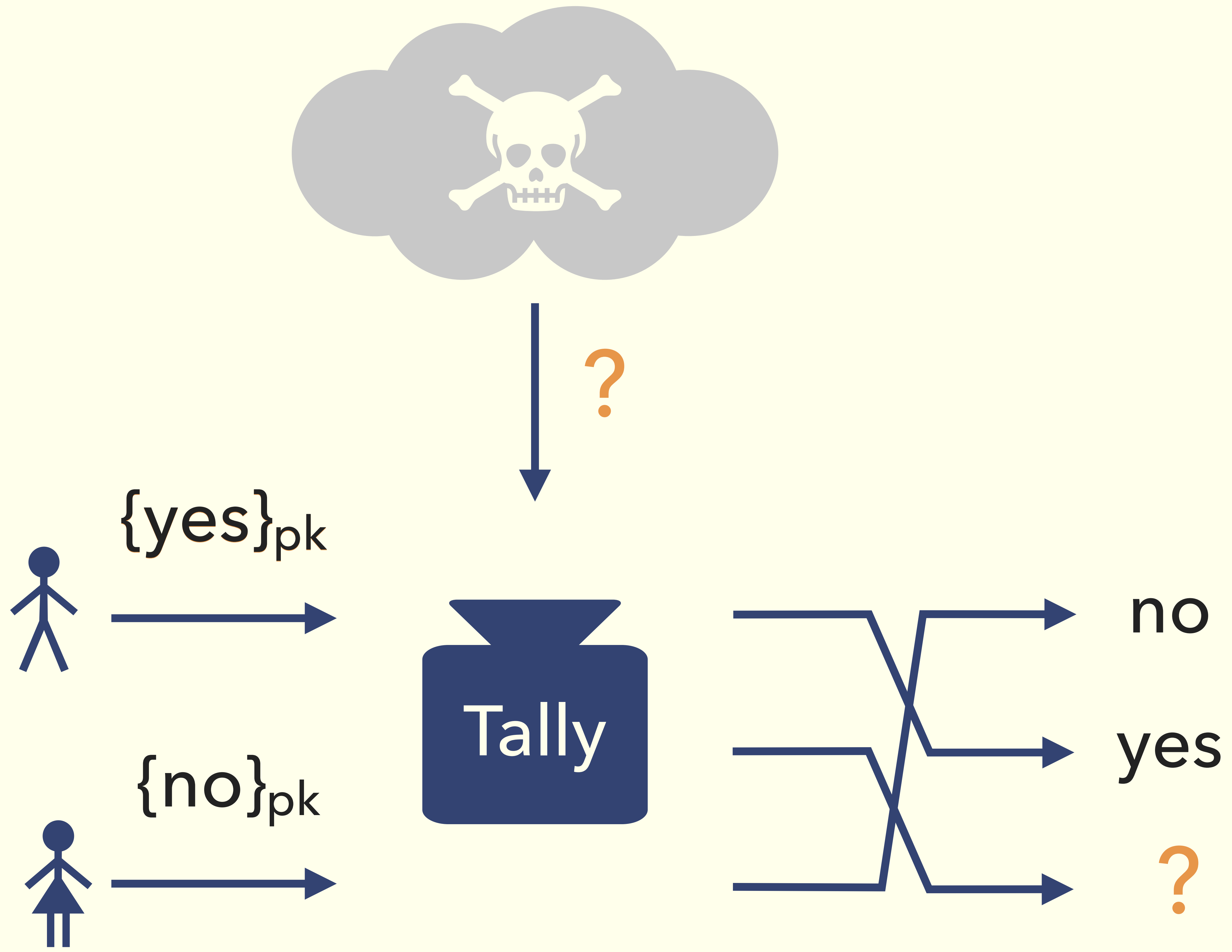


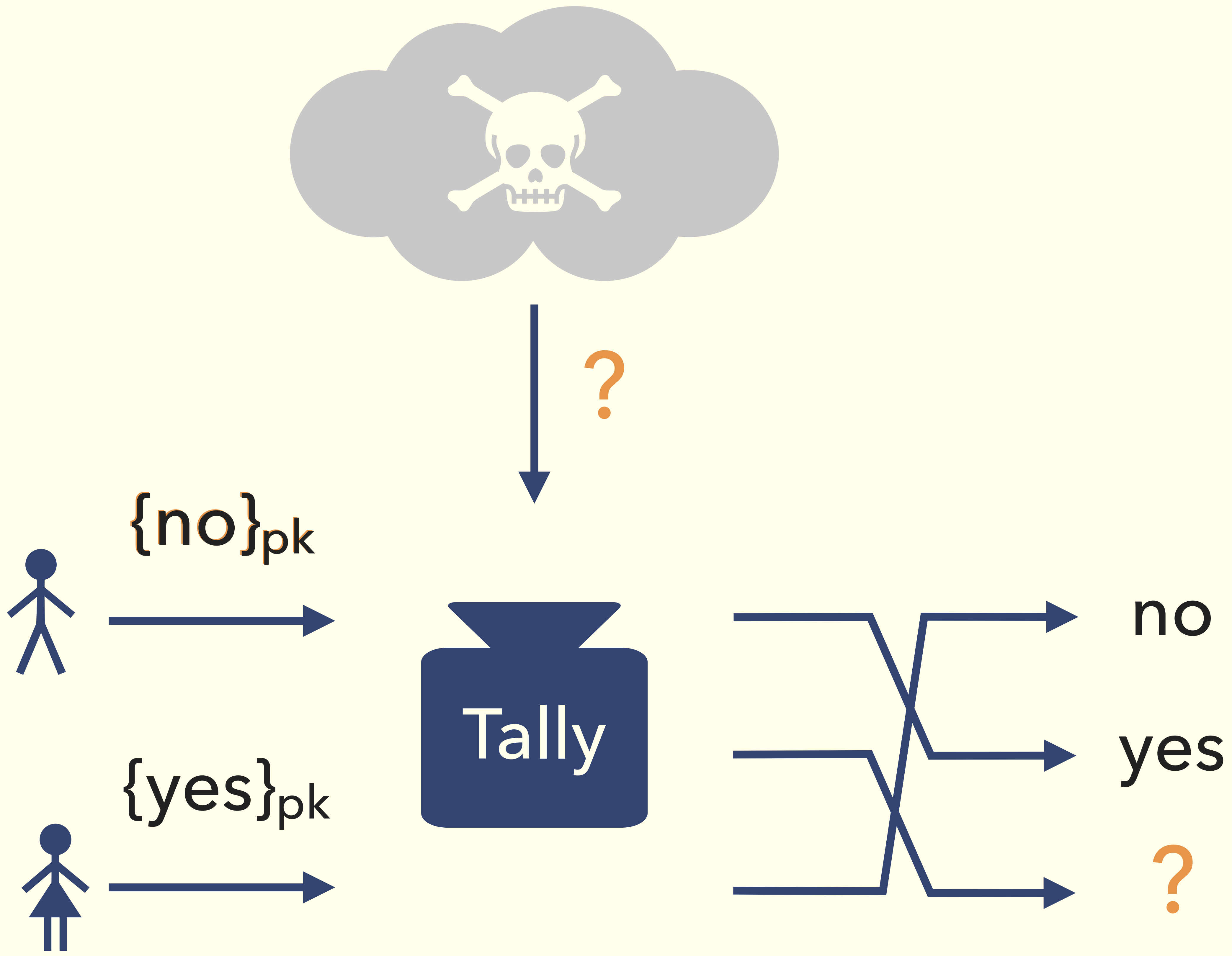


A simple e-voting protocol

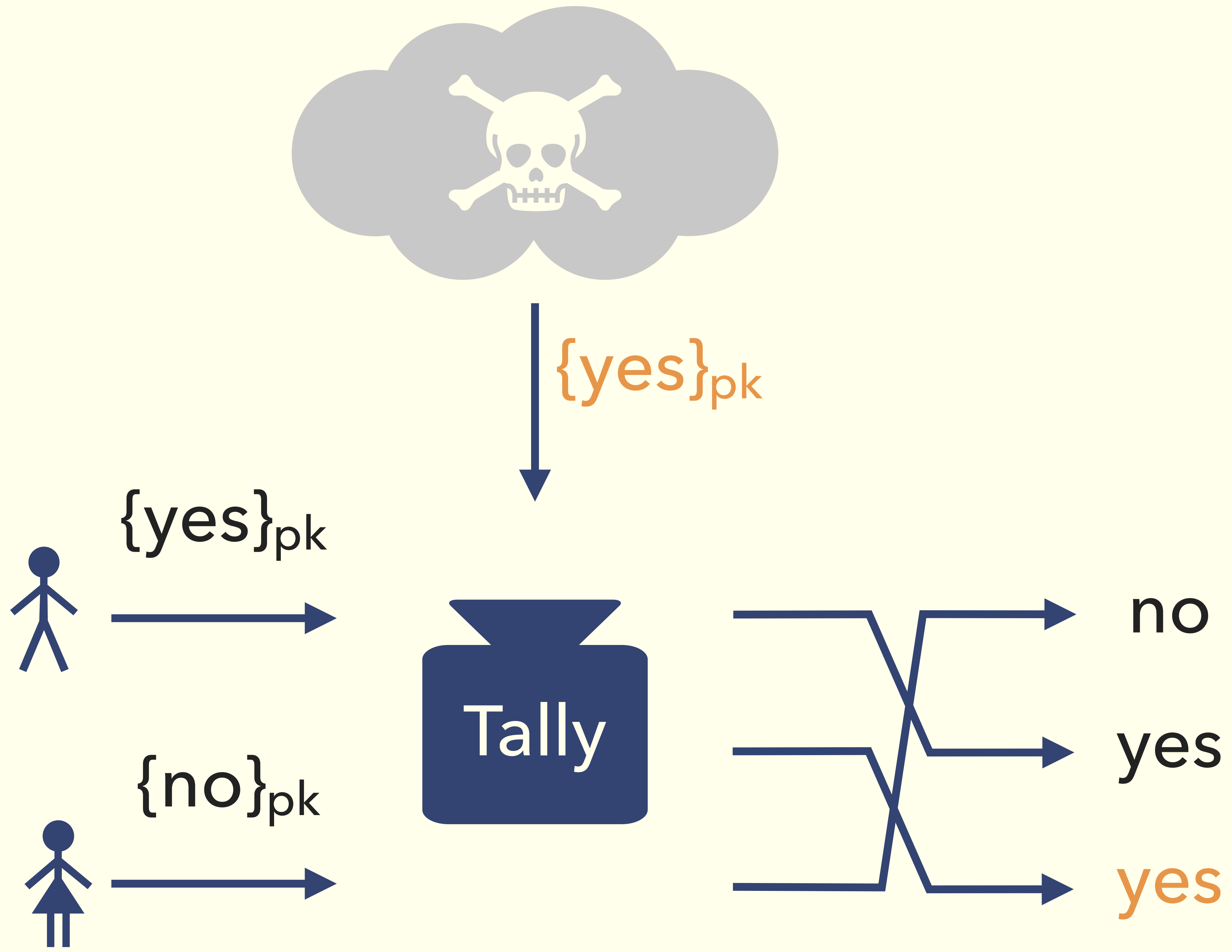


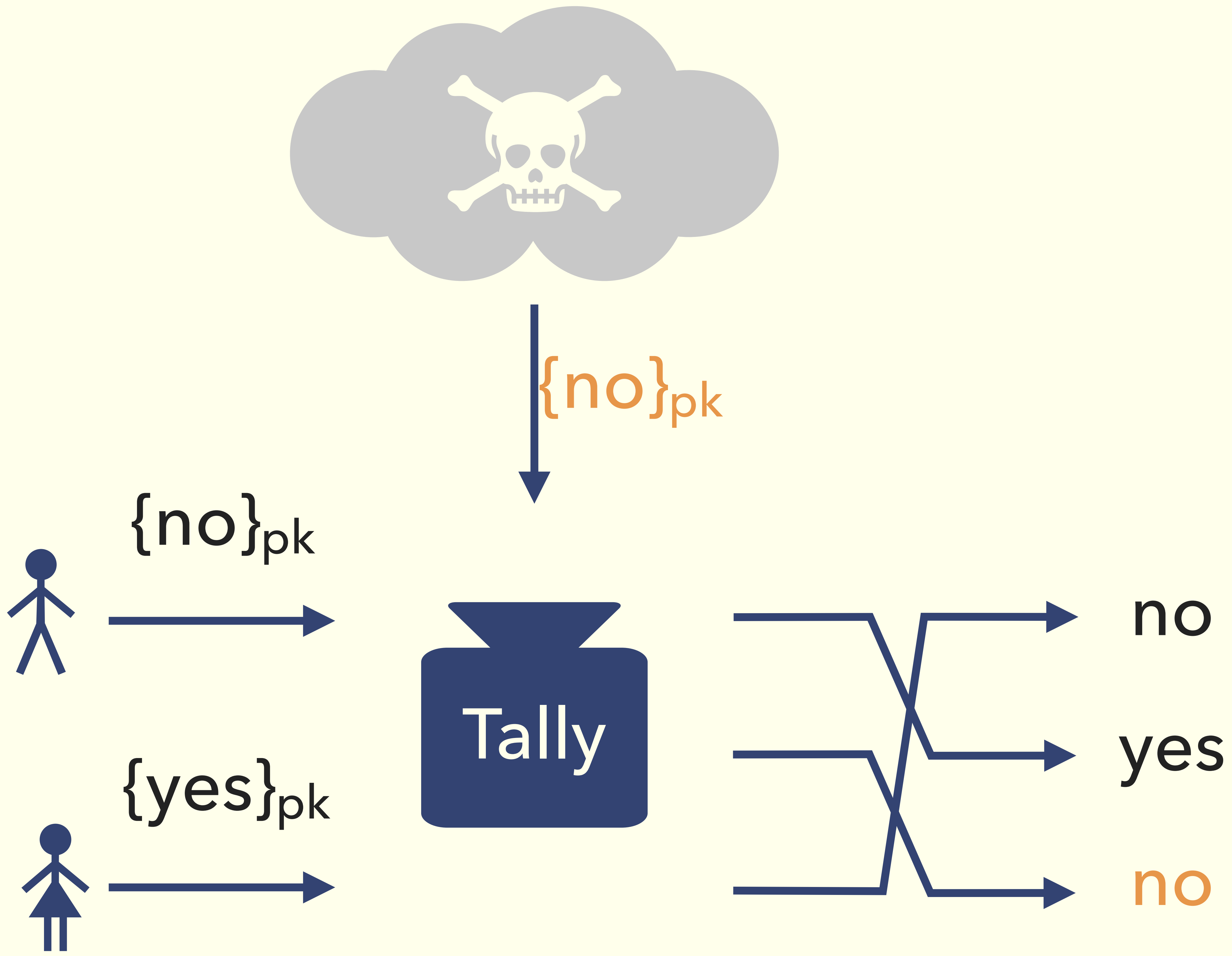
A simple e-voting protocol



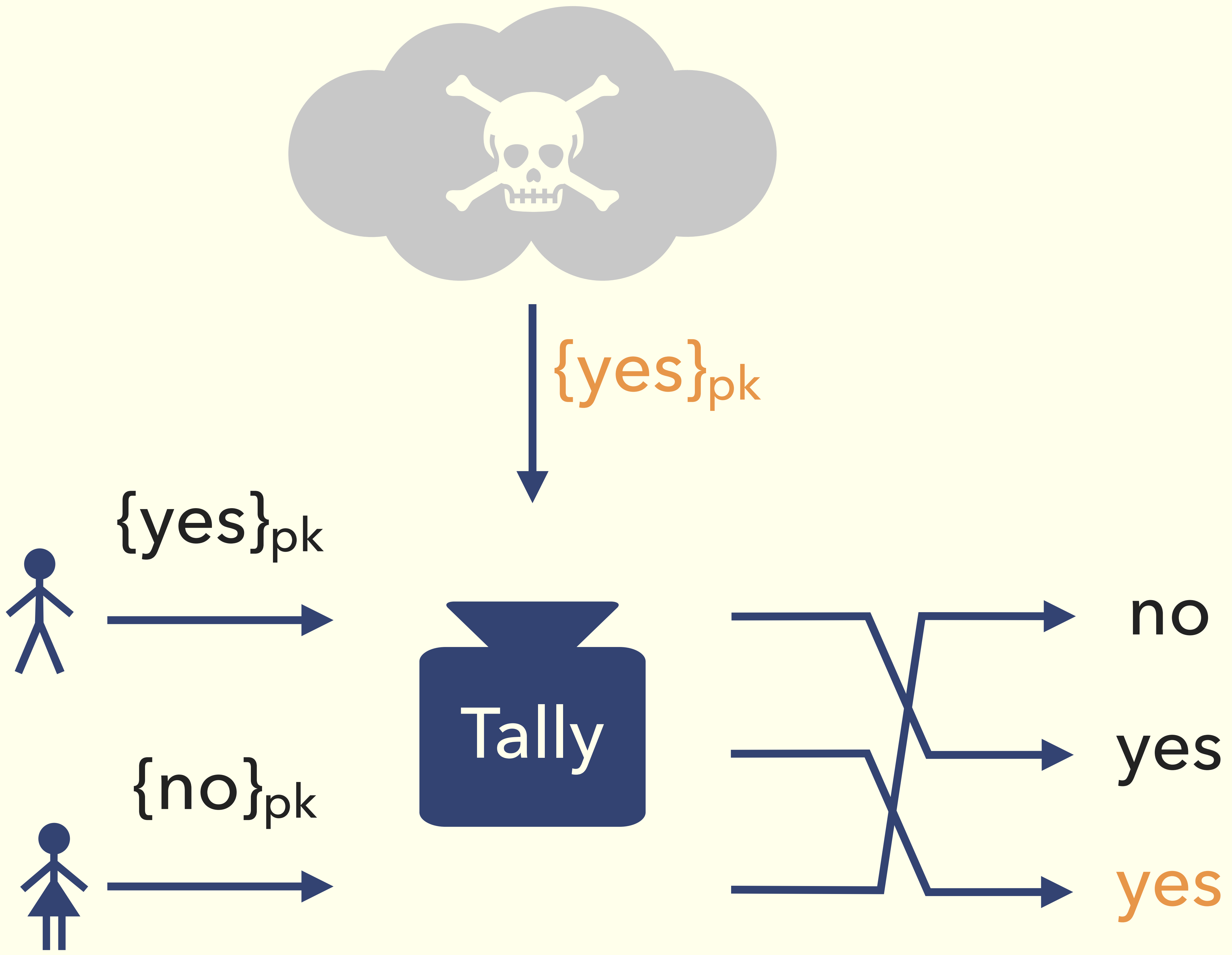


A simple e-voting protocol

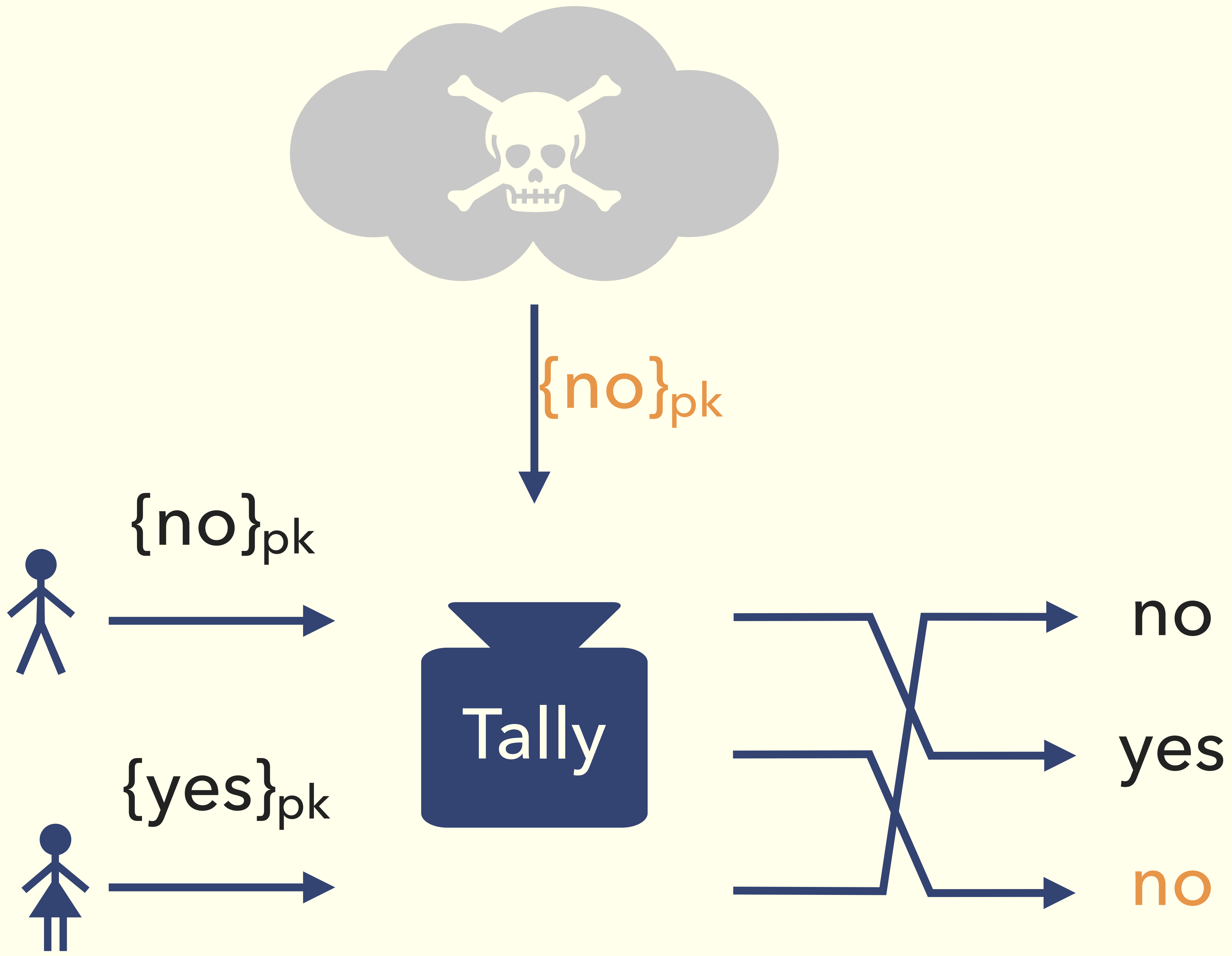




A simple e-voting protocol

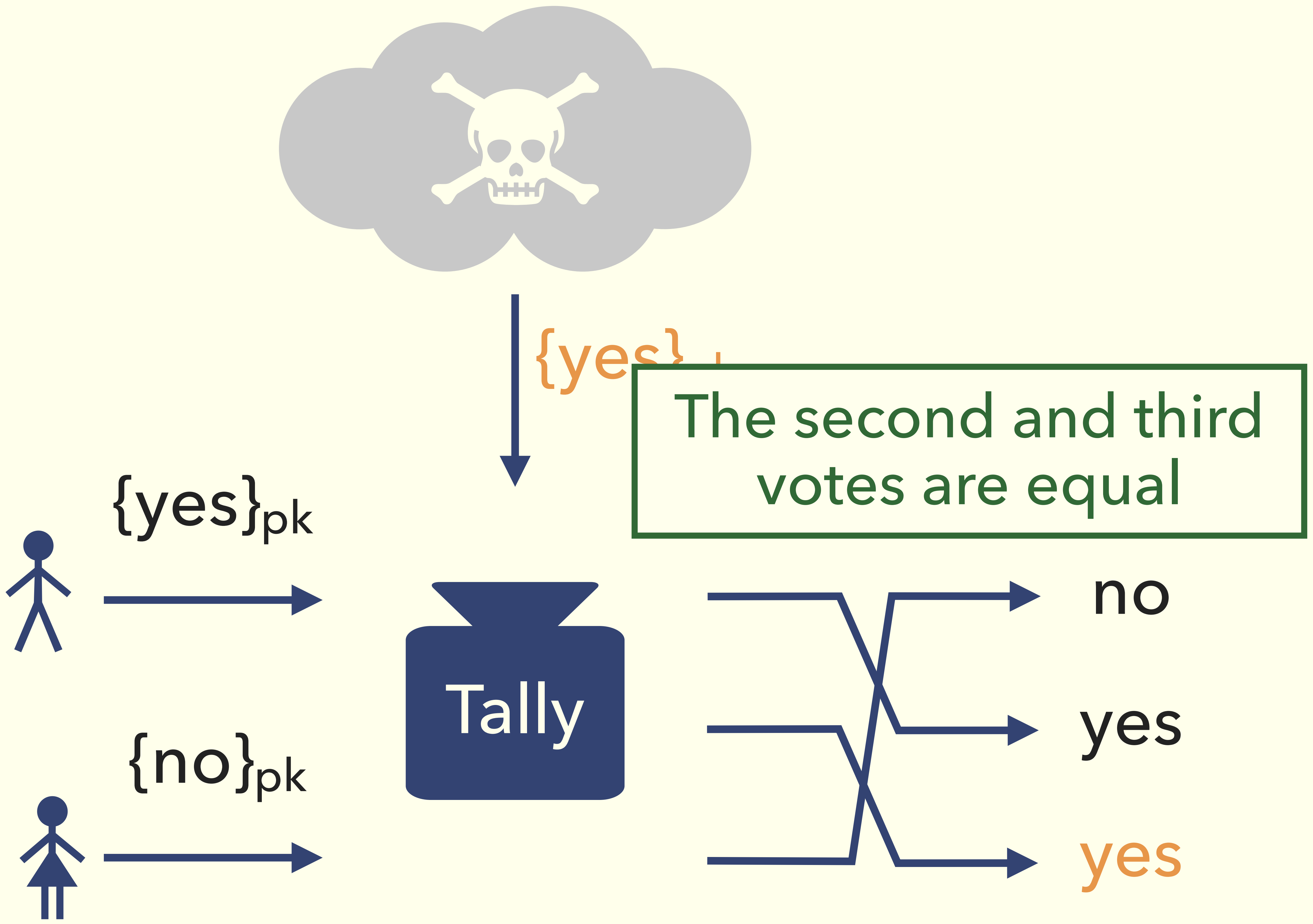


the vote appearing twice on the bulletin board is Bob's vote!

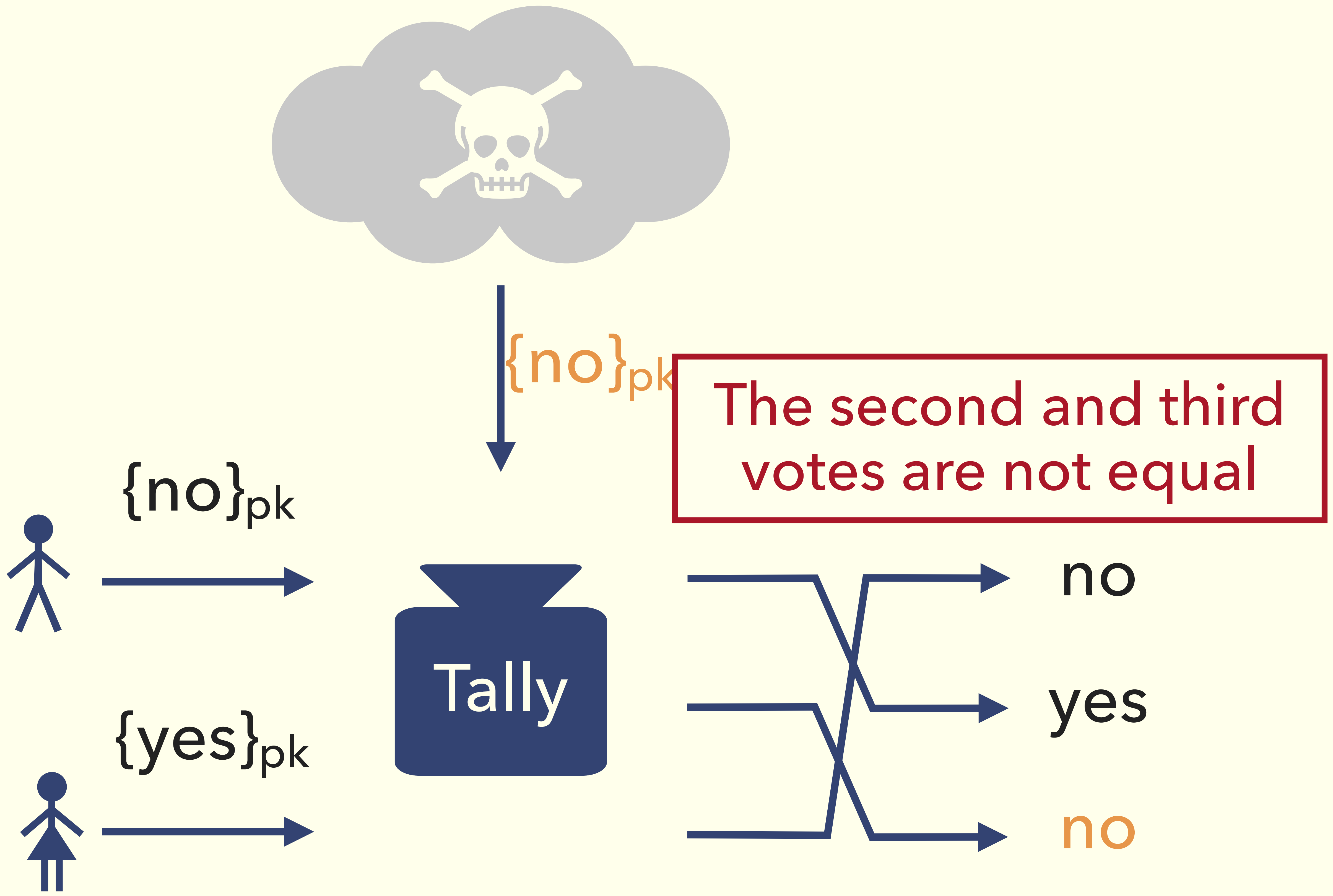


the vote appearing twice on the bulletin board is Bob's vote!

A simple e-voting protocol

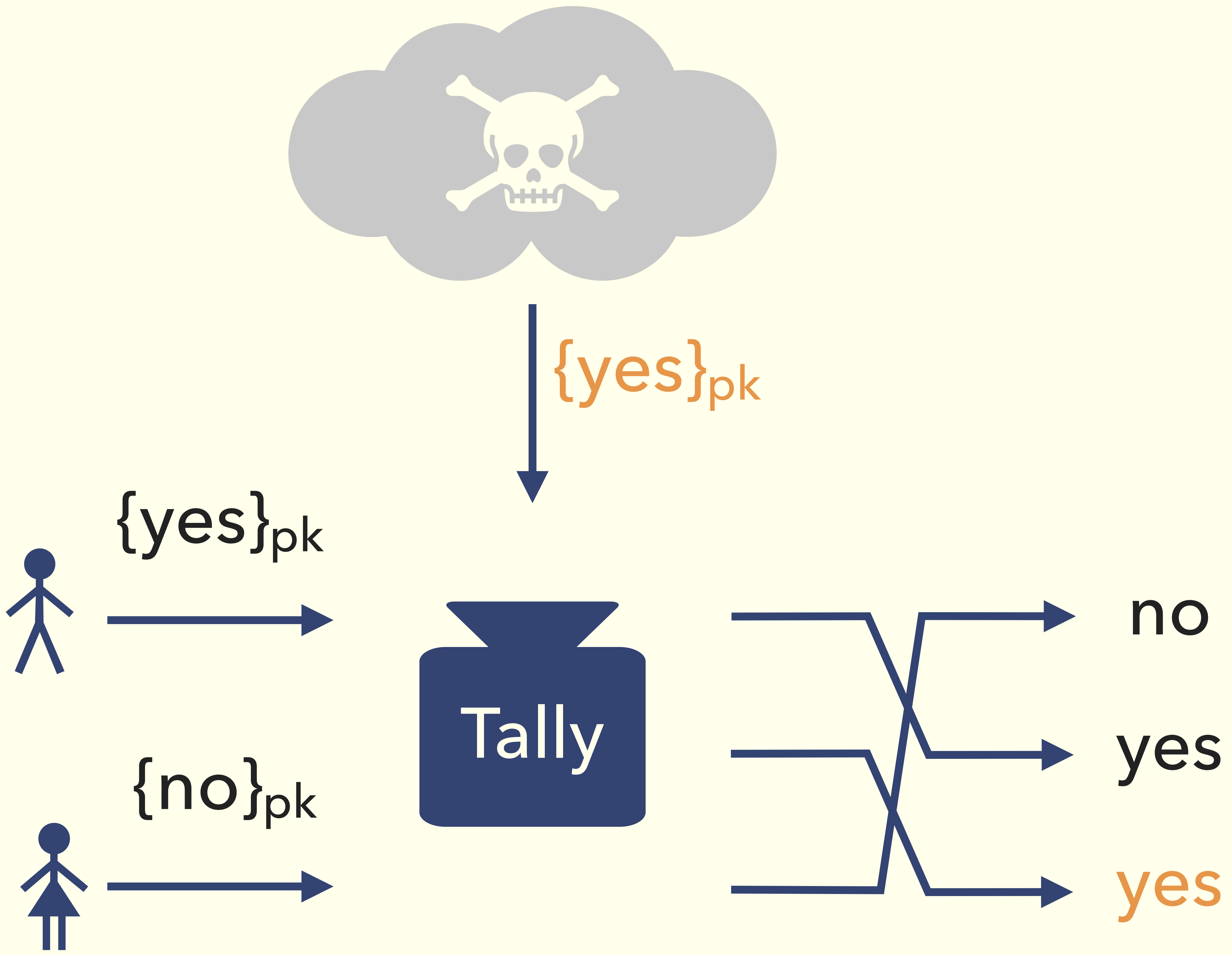


the vote appearing twice on the bulletin board is Bob's vote!

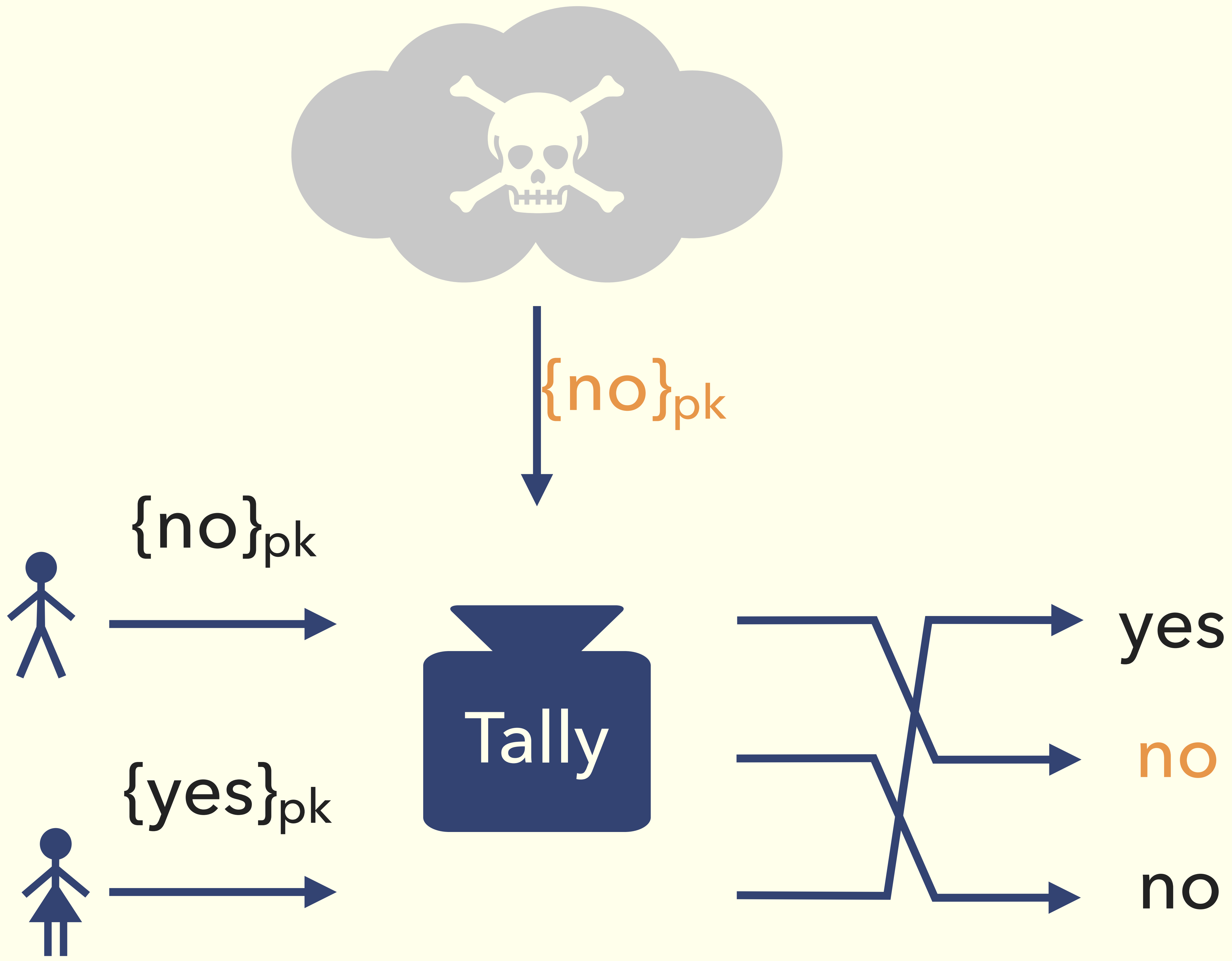


the vote appearing twice on the bulletin board is Bob's vote!

A simple e-voting protocol

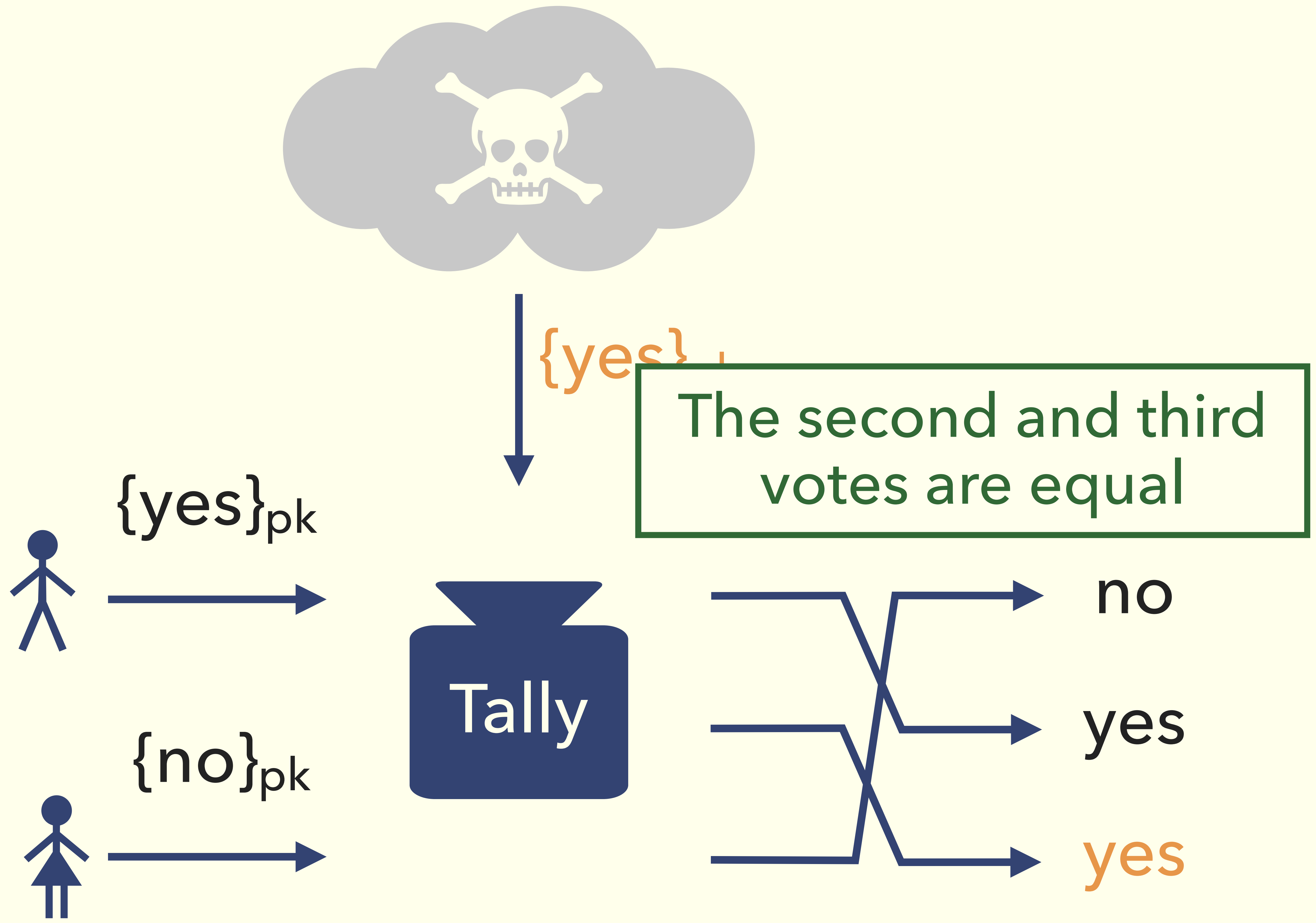


the vote appearing twice on the bulletin board is Bob's vote!

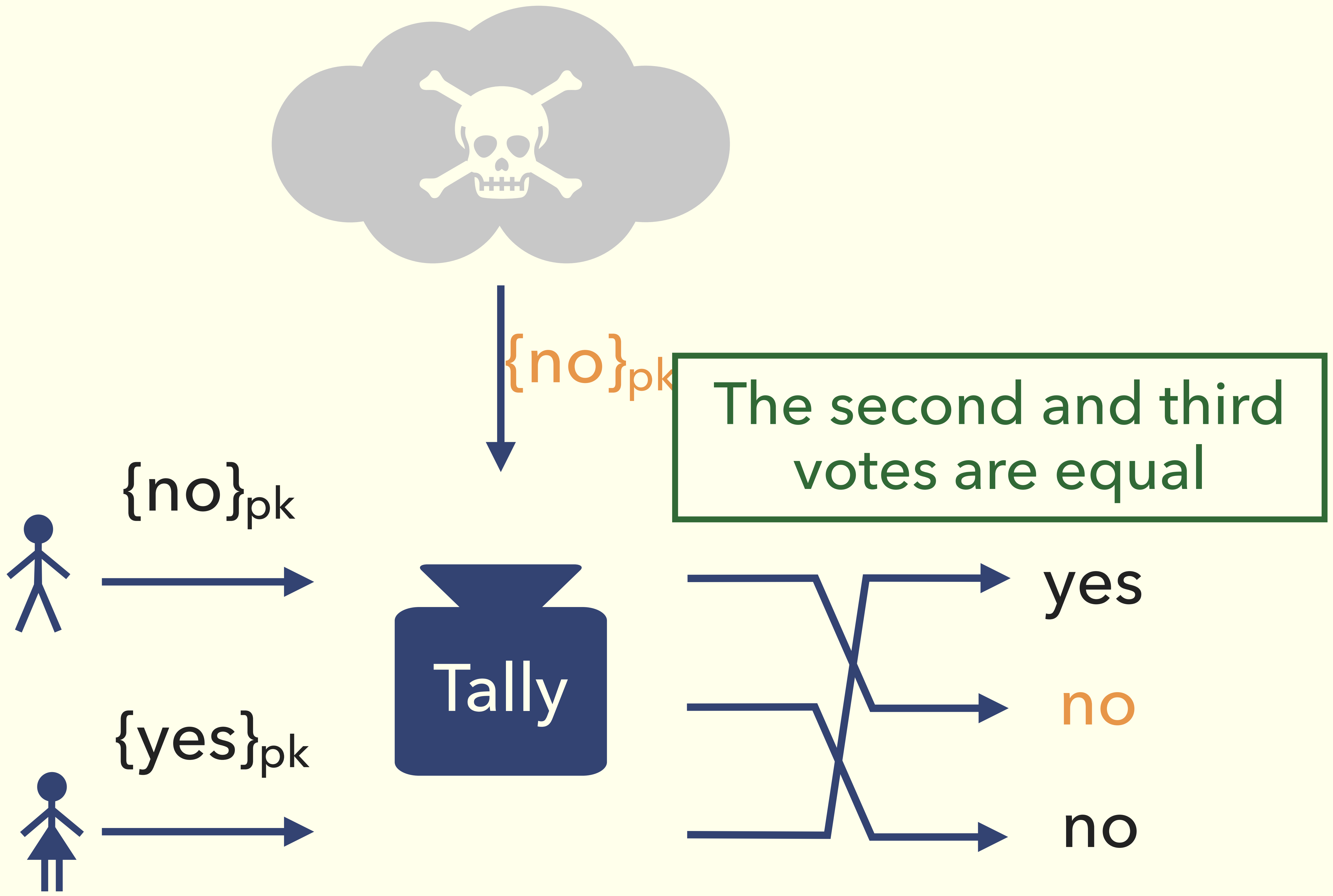


the vote appearing twice on the bulletin board is Bob's vote!

A simple e-voting protocol

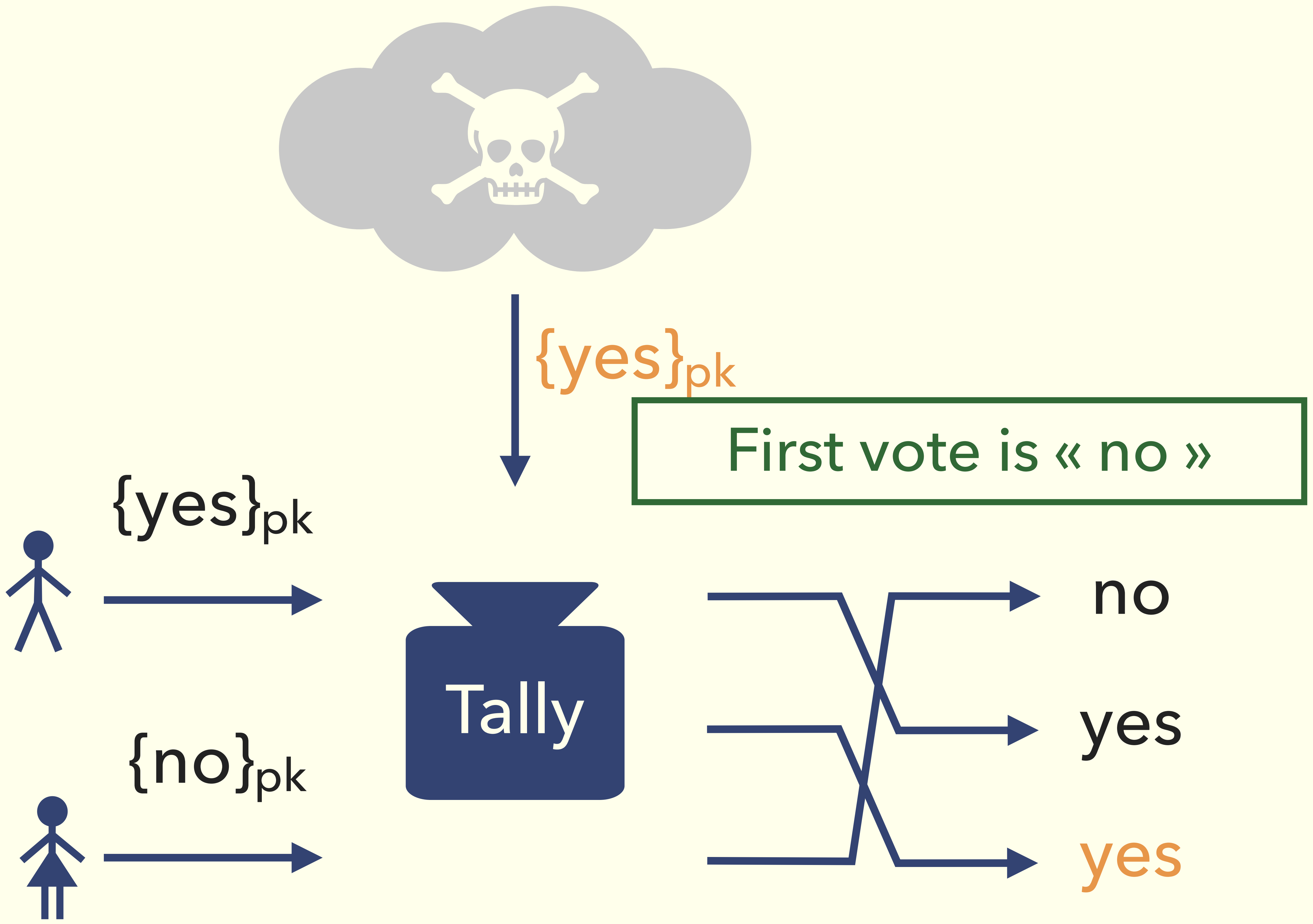


the vote appearing twice on the bulletin board is Bob's vote!

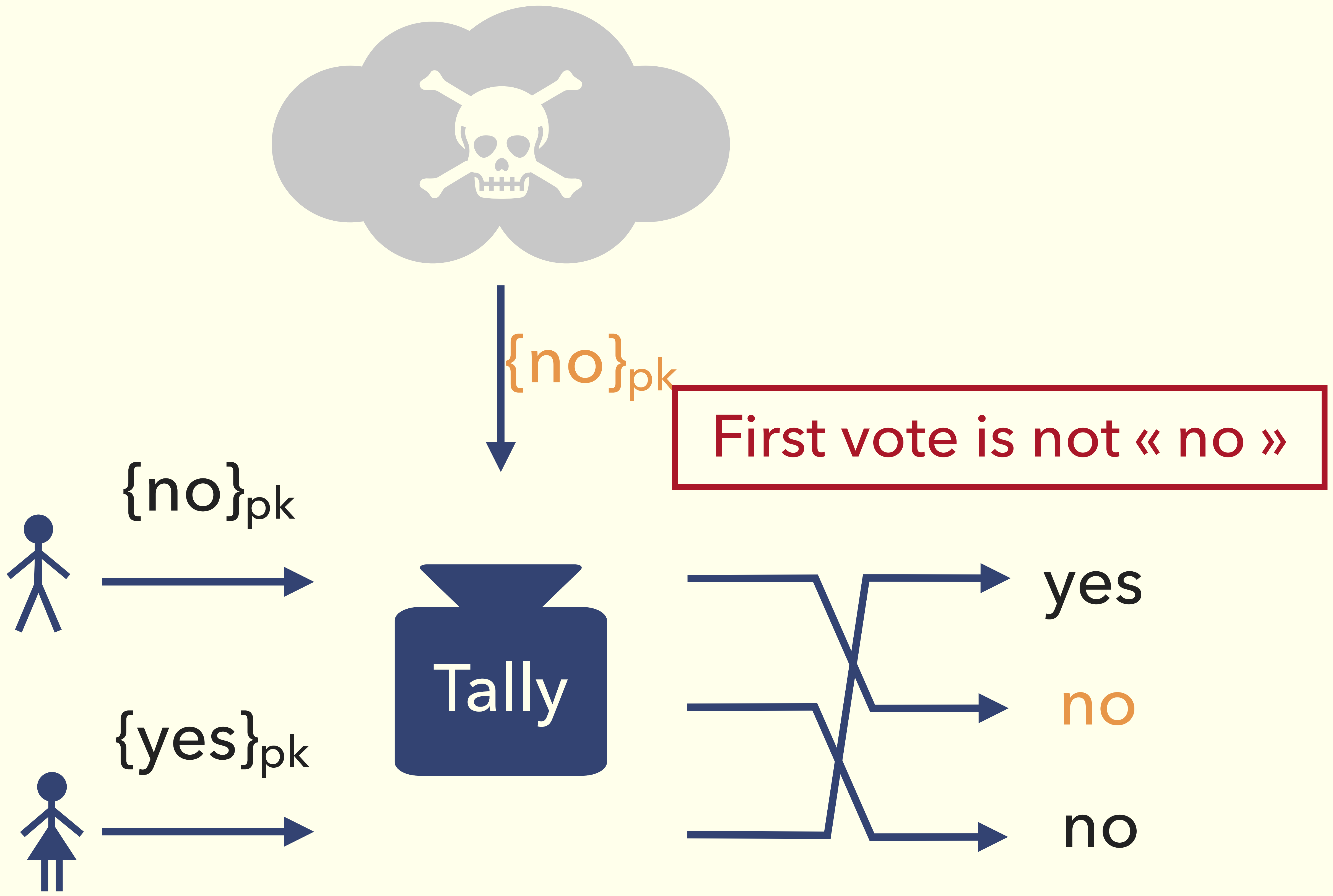


the vote appearing twice on the bulletin board is Bob's vote!

A simple e-voting protocol



the vote appearing twice on the
bulletin board is Bob's vote!



the vote appearing twice on the
bulletin board is Bob's vote!

Equivalence of processes in ProVerif

```
let system1 = setup | voter(skA,v1) | voter(skB,v2).
let system2 = setup | voter(skA,v2) | voter(skB,v1).

equivalence system1 system2
```

Equivalence between two processes

```
let system(vA,vB) = setup | voter(skA,vA) | voter(skB,vB).

process system(choice[v1,v2],choice[v2,v1])
```

Equivalence as a biprocess

Equivalence of processes in ProVerif

```
let system1 = setup | voter(skA,v1) | voter(skB,v2).
let system2 = setup | voter(skA,v2) | voter(skB,v1).

equivalence system1 system2
```

Equivalence between two processes

Internally

Equivalence as a biprocess

```
let system(vA,vB) = setup | voter(skA,vA) | voter(skB,vB).

process system(choice[v1,v2],choice[v2,v1])
```

Equivalence of processes in ProVerif

Equivalence between two processes

+ Easier to model,
+ No need to know « how to match » the processes
- Can be slow
- Difficult to « fix » when not working

Equivalence as a biprocess

+ Also easy to model,
+ Works better with other features (e.g. lemmas, axioms)
+ More efficient
• Need to have a good idea why processes are equivalent

Memory cell

ProVerif’s calculus is stateless

Memory cell

ProVerif’s calculus is stateless … but we have private channels

Memory cell

ProVerif's calculus is stateless … but we have private channels

A Ocaml like version

let x = ref 0	Initialisation
!x	Reading
x := n	Writing

Memory cell

ProVerif's calculus is stateless … but we have private channels

A Ocaml like version

let x = ref 0 Initialisation

```
free cell:channel [private]
let init = out(cell,0).
```

!x Reading

x := n Writing

Memory cell

ProVerif's calculus is stateless … but we have private channels

A Ocaml like version

let x = ref 0 Initialisation

```
free cell:channel [private]
let init = out(cell,0).
```

!x Reading

```
let P =
  …
  in(cell,x:nat); out(cell,x);
  …
```

x := n Writing

Memory cell

ProVerif's calculus is stateless … but we have private channels

A Ocaml like version

let x = ref 0 Initialisation

```
free cell:channel [private]
let init = out(cell,0).
```

!x Reading

```
let P =
  …
  in(cell,x:nat); out(cell,x);
  …
```

x := n Writing

```
let Q =
  …
  in(cell,x:nat); out(cell,n);
  …
```

Memory cell

Initialisation

```
free cell:channel [private]
let init = out(cell,0).
```

Reading

```
let P =
  …
  in(cell,x:nat); out(cell,x);
  …
```

Writing

```
let Q =
  …
  in(cell,x:nat); out(cell,n);
  …
```

The system

```
process
  init | P | Q | !in(cell,x:nat);out(cell,x)
```

Avoids « blocking » an agent

Reading/Writing both consist of inputing the « current value » of the cell and outputting the « new value »

Communication are synchronous on private channels: always one single output available at all time.

Locking memory cell

Initialisation

```
free cell:channel [private]
let init = out(cell,0).
```

Reading

```
let P =
  …
  in(cell,x:nat);
  event B;
  out(cell,x);
  …
```

Writing

```
let Q =
  …
  in(cell,x:nat);
  event A;
  event C;
  out(cell,n);
  …
```

Communication are synchronous on private channels:
If no output available, all processes trying to input are « blocked »

The sequence of events A, B, C is not possible

Locking memory cell

Initialisation

```
free cell:channel [private]
let init = out(cell,0).
```

Lock and read

```
let P =
  …
  in(cell,x:nat);
  event B;
  out(cell,x);
  …
```

Write and unlock

```
let Q =
  …
  in(cell,x:nat);
  event A;
  event C;
  out(cell,n);
  …
```

Communication are synchronous on private channels:
If no output available, all processes trying to input are « blocked »

The sequence of events A, B, C is not possible

Simplified Yubikey protocol

P only accepts increasing sequence of natural numbers.

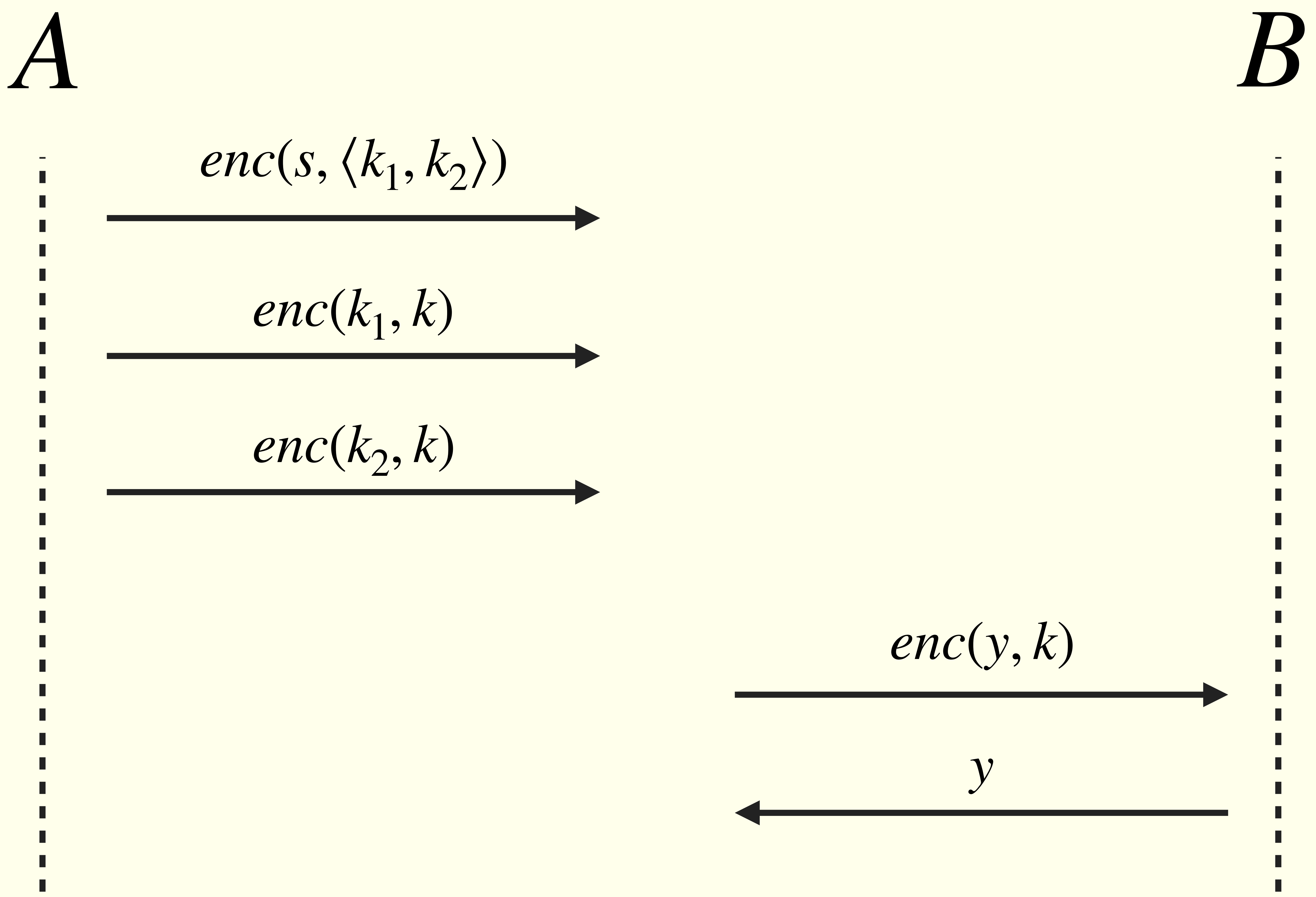
Q emits sequentially
all natural numbers encrypted
with k

```
free k:key [private].
free cellP,cellQ:channel [private]

let P =
 in(c,x:bitstring);
 in(cellP,i:nat);
 let j = sdec(x,k) in
 if j > i
 then
  event Accept(j);
  out(cellP,j)
 else
  out(cellP,i).

let Q =
 in(cellQ,i:nat);
 out(c,senc(I,k));
 out(cellQ,i+1).

process out(cellP,0) | out(cellQ,0) | !P | !Q
```

Outline

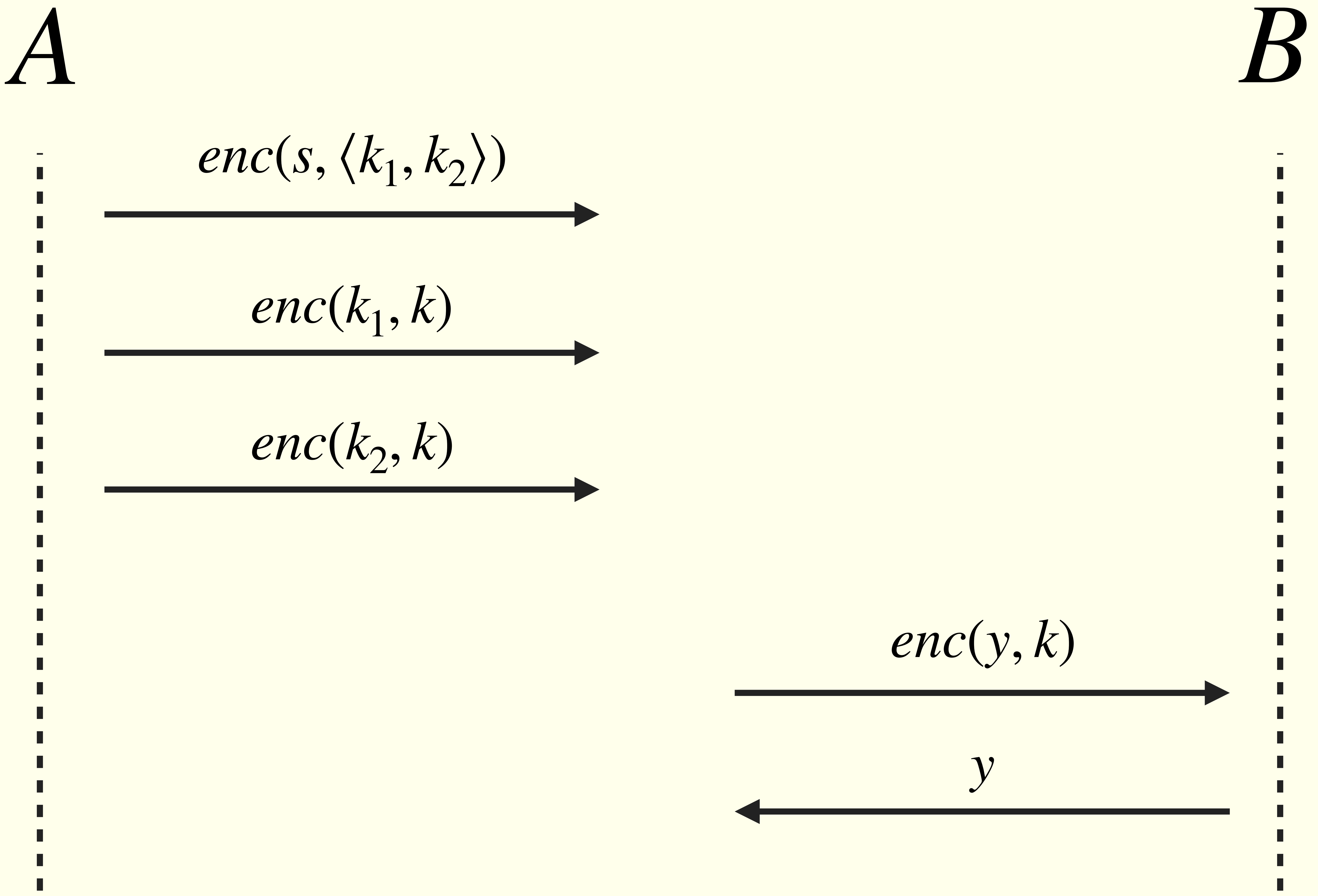
A. Additional modeling techniques
   1. Equational Theory vs Rewrite rules
   2. Equivalence properties
   3. Memory cell and locks for stateful protocols
B. Dealing with « cannot be proved »
   1. Adding precision
   2. Trace with assumption
   3. Proof by induction
   4. Lemma
C. Dealing with non-termination

Toy-example



Toy-example

B acts as an oracle for decryption with the key k
but only one time !



Why does it fail ?

Transform process in Horn clauses

```
free s,k1,k2,k:bitstring [private].

let A =
 out(c,senc(s,(k1,k2)));
 out(c,senc(k1,k));
 out(c,senc(k2,k)).

let B =
 in(c,x);
 out(c,dec(x,k).

process A | B
```

$$\rightarrow att(\mathsf{enc}(s, \langle k_1, k_2\rangle))$$

$$\rightarrow att(\mathsf{enc}(k_1, k))$$

$$\rightarrow att(\mathsf{enc}(k_2, k))$$

$$att(\mathsf{enc}(y, k)) \rightarrow att(y)$$

Horn clauses for the attacker

$$att(x) \wedge att(y) \rightarrow att(\mathsf{enc}(x, y))$$

$$att(enc(x, y)) \wedge att(y) \rightarrow att(x)$$

$$att(x) \wedge att(y) \rightarrow att(\langle x, y\rangle)$$

Why does it fail ?

Transform process in Horn clauses

```
free s,k1,k2,k:bitstring [private].

let A =
 out(c,senc(s,(k1,k2)));
 out(c,senc(k1,k));
 out(c,senc(k2,k)).

let B =
 in(c,x);
 out(c,dec(x,k).

process A | B
```

$$\rightarrow att(\mathsf{enc}(s, \langle k_1, k_2 \rangle))$$

$$\rightarrow att(\mathsf{enc}(k_1, k))$$

$$\rightarrow att(\mathsf{enc}(k_2, k))$$

$$att(\mathsf{enc}(y, k)) \rightarrow att(y)$$

Horn clauses for the attacker

$$att(x) \wedge att(y) \rightarrow att(\mathsf{enc}(x, y))$$

$$att(enc(x, y)) \wedge att(y) \rightarrow att(x)$$

$$att(x) \wedge att(y) \rightarrow att(\langle x, y \rangle)$$

Secrecy of *s* is preserved if *att(s)* is not logically decucible from the set of Horn clauses

Why does it fail ?

Transform process in Horn clauses

```
free s,k1,k2,k:bitstring [private].

let A =
 out(c,senc(s,(k1,k2)));
 out(c,senc(k1,k));
 out(c,senc(k2,k)).

let B =
 in(c,x);
 out(c,dec(x,k).

process A | B
```

$$\rightarrow att(\mathsf{enc}(s, \langle k_1, k_2\rangle))$$

$$\rightarrow att(\mathsf{enc}(k_1, k))$$

$$\rightarrow att(\mathsf{enc}(k_2, k))$$

$$att(\mathsf{enc}(y, k)) \rightarrow att(y)$$

Horn clauses can be applied an arbitrary number of times for arbitrary instanciations

Horn clauses for the attacker

$$att(x) \wedge att(y) -$$

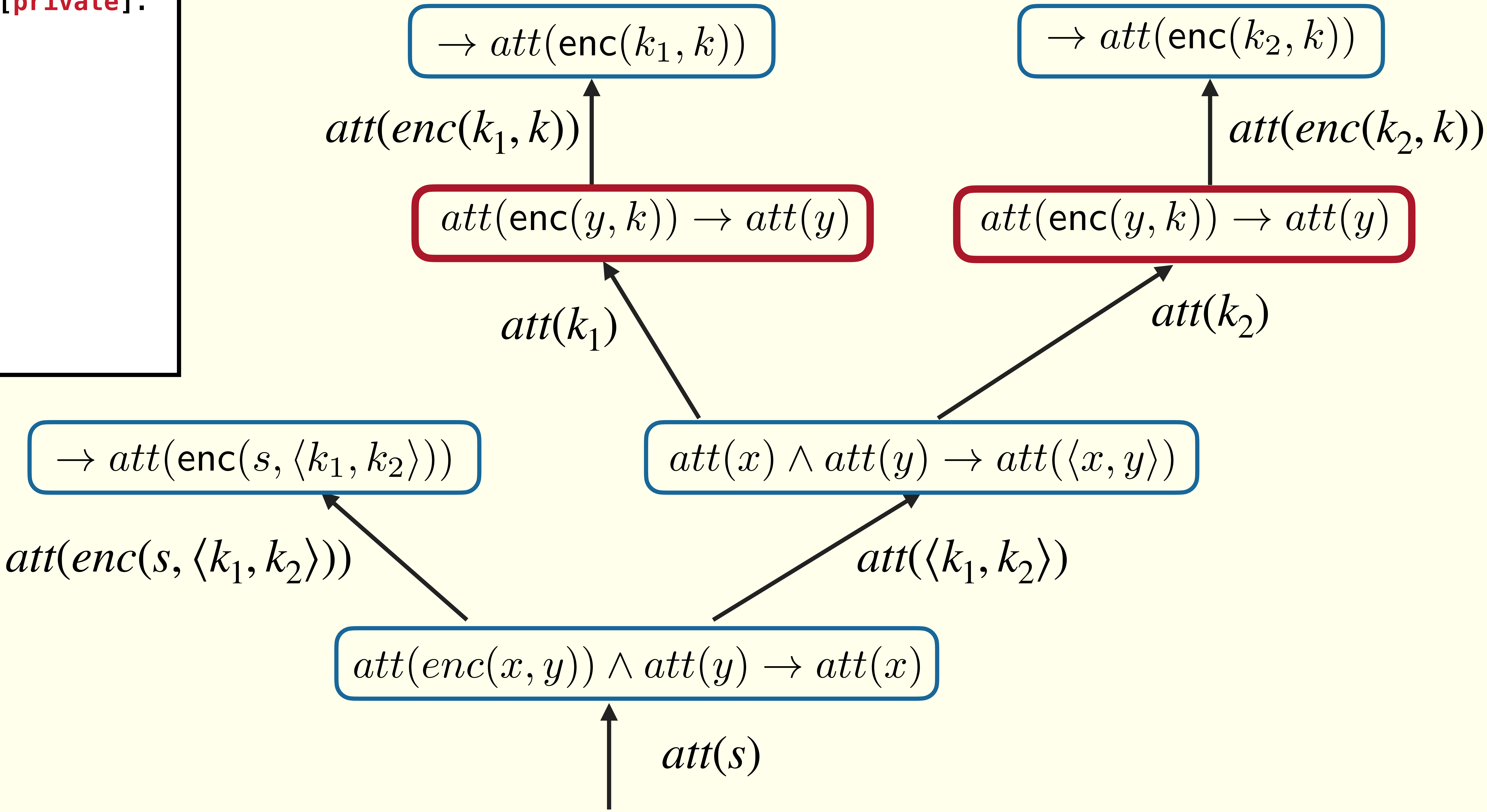
$$att(enc(x, y)) \wedge att(y) -$$

$$att(x) \wedge att(y) \rightarrow att(\langle x, y\rangle)$$

Secrecy of *s* is preserved if *att(s)* is not logically decucible from the set of Horn clauses

Why does it fail ?

```
free s,k1,k2,k:bitstring [private].

let A =
 out(c,senc(s,(k1,k2)));
 out(c,senc(k1,k));
 out(c,senc(k2,k)).

let B =
 in(c,x);
 out(c,dec(x,k).

process A | B
```



What to do ?

Add a [precise] option to the problematic input !

```
free s,k1,k2,k:bitstring [private].

let A =
 out(c,senc(s,(k1,k2)));
 out(c,senc(k1,k));
 out(c,senc(k2,k)).

let B =
 in(c,x) [precise];
 out(c,dec(x,k).

process A | B
```

```
-- Query not attacker(s[]) in process 0.
Translating the process into Horn clauses...
Completing...
Starting query not attacker(s[])

RESULT not attacker(s[]) is true.

--------------------------------------------------------------
Verification summary:

Query not attacker(s[]) is true.

--------------------------------------------------------------
```

Global setting

```
set preciseActions = true.
```

⚠ Adding [precise] options may increase the verification time or lead to non-termination

How to know where to put precise ?

Going through the derivation !

Find two different messages received by the same input {n}

Check on your process if it should be possible

```
Derivation:

1. The message enc(k2[],k[]) may be sent to the attacker at output {5}.
attacker(enc(k2[],k[])).

2. The message enc(k2[],k[]) that the attacker may have by 1 may be received at input {7}.
So the message k2[] may be sent to the attacker at output {8}.
attacker(k2[]).

3. The message enc(k1[],k[]) may be sent to the attacker at output {4}.
attacker(enc(k1[],k[])).

4. The message enc(k1[],k[]) that the attacker may have by 3 may be received at input {7}.
So the message k1[] may be sent to the attacker at output {8}.
attacker(k1[]).

5. By 4, the attacker may know k1[].
By 2, the attacker may know k2[].
Using the function 2-tuple the attacker may obtain (k1[],k2[]).
attacker((k1[],k2[])).

6. The message enc(s[],(k1[],k2[])) may be sent to the attacker at output {6}.
attacker(enc(s[],(k1[],k2[]))).

7. By 6, the attacker may know enc(s[],(k1[],k2[])).
By 5, the attacker may know (k1[],k2[]).
Using the function dec the attacker may obtain s[].
attacker(s[]).

8. By 7, attacker(s[]).
The goal is reached, represented in the following fact:
attacker(s[]).
```

How to know where to put precise ?

Going through the derivation !

Find two different messages received by the same input {n}

Check on your process if it should be possible

```
Derivation:

1. The message enc(k2[],k[]) may be sent to the attacker at output {5}.
attacker(enc(k2[],k[])).

2. The message enc(k2[],k[]) that the attacker may have by 1 may be received at input {7}.
So the message k2[] may be sent to the attacker at output {8}.
attacker(k2[]).

3. The message enc(k1[],k[]) may be sent to the attacker at output {4}.
attacker(enc(k1[],k[])).

4. The message enc(k1[],k[]) that the attacker may have by 3 may be received at input {7}.
So the message k1[] may be sent to the attacker at output {8}.
attacker(k1[]).

5. By 4, the attacker may know k1[].
By 2, the attacker may know k2[].
Using the function 2-tuple the attacker may obtain (k1[],k2[]).
attacker((k1[],k2[])).

6. The message enc(s[],(k1[],k2[])) may be sent to the attacker at output {6}.
attacker(enc(s[],(k1[],k2[]))).

7. By 6, the attacker may know enc(s[],(k1[],k2[])).
By 5, the attacker may know (k1[],k2[]).
Using the function dec the attacker may obtain s[].
attacker(s[]).

8. By 7, attacker(s[]).
The goal is reached, represented in the following fact:
attacker(s[]).
```

Two strange situations !

Simplified Yubikey

```
free k:key [private].
free cellP,cellQ:channel [private]

let P =
 in(c,x:bitstring);
 in(cellP,i:nat);
 let j = sdec(x,k) in
 if j > i
 then
  event Accept(j);
  out(cellP,j)
 else
  out(cellP,i).

let Q =
 in(cellQ,i:nat);
 out(c,senc(I,k));
 out(cellQ,i+1).

process out(cellP,0) | out(cellQ,0) | !P | !Q
```

Can't disprove the sanity check…

```
query i:nat; event(Accept(i)).
```

```
-- Query not event(Accept(i_2)) in process 0.
Translating the process into Horn clauses...
mess(cellQ[],i_2) -> mess(cellQ[],i_2 + 1)
select mess(cellQ[],i_2)/-5000
Completing...
Starting query not event(Accept(i_2))

goal reachable: i_2 ≥ 1 && mess(cellQ[],i_2) -> end(Accept(i_2))

Derivation:

1. We assume as hypothesis that
mess(cellQ[],i_2).

2. The message i_2 that may be sent on channel cellQ[] by 1 may be received
at input {12}.
So the message senc(i_2,k[]) may be sent to the attacker at output {13}.
attacker(senc(i_2,k[])).

…

Could not find a trace corresponding to this derivation.
```

Two strange situations !

```
-- Query not attacker(S(kAminus[!1 = v],x_1)) in process 1.
select member(*x_1,y)/-5000
select memberid(*x_1,y)/-5000
Translating the process into Horn clauses...
Completing...

…

A more detailed output of the traces is available with
  set traceDisplay = long.

new exponent: channel creating exponent_3 at {1}

new honestC: channel creating honestC_3 at {8}

new kAminus: skey creating kAminus_3 at {10} in copy a

…
…

event enddosi(Pk(kAminus_3),NI_3) at {37} in copy a, a_4, a_8

event mess3(Pk(kAminus_3)…))) at {46} in copy a, a_4, a_8

out(c, cons3(~M_9,….)) at {47} in copy a, a_4, a_8

The attacker has the message 3-proj-3-tuple(D(H(…)).
A trace has been found, assuming the following hypothesis:
memberid(Pk(a_12[]),a_5[])
Stopping attack reconstruction attempts. To try more traces,
modify the setting reconstructTrace.
RESULT not attacker(S(kAminus[!1 = v],x_1)) cannot be proved.
```

A trace is found… but ProVerif assume that the attacker has magically a term

A closer look

```
-- Query not attacker(S(kAminus[!1 = v],x_1)) in process 1.
select member(*x_1,y)/-5000
select memberid(*x_1,y)/-5000
Translating the process into Horn clauses...
Completing...

…

A more detailed output of the traces is available with
  set traceDisplay = long.

new exponent: channel creating exponent_3 at {1}

new honestC: channel creating honestC_3 at {8}

new kAminus: skey creating kAminus_3 at {10} in copy a

…
…

event enddosi(Pk(kAminus_3),NI_3) at {37} in copy a, a_4, a_8

event mess3(Pk(kAminus_3)…))) at {46} in copy a, a_4, a_8

out(c, cons3(~M_9,….)) at {47} in copy a, a_4, a_8

The attacker has the message 3-proj-3-tuple(D(H(…)).
A trace has been found, assuming the following hypothesis:
memberid(Pk(a_12[]),a_5[])
Stopping attack reconstruction attempts. To try more traces,
modify the setting reconstructTrace.
RESULT not attacker(S(kAminus[!1 = v],x_1)) cannot be proved.
```

```
-- Query not event(Accept(i_2)) in process 0.
Translating the process into Horn clauses...
mess(cellQ[],i_2) -> mess(cellQ[],i_2 + 1)
select mess(cellQ[],i_2)/-5000
Completing...
Starting query not event(Accept(i_2))

goal reachable: i_2 ≥ 1 && mess(cellQ[],i_2) ->
end(Accept(i_2))

Derivation:

1. We assume as hypothesis that
mess(cellQ[],i_2).

2. The message i_2 that may be sent on channel cellQ[] by 1 may
be received at input {12}.
So the message senc(i_2,k[]) may be sent to the attacker at
output {13}.
attacker(senc(i_2,k[])).

…

Could not find a trace corresponding to this derivation.
```

ProVerif decided to prevent resolution on some facts

Why ProVerif prevent resolution ?

Simplified Yubikey

```
free k:key [private].
free cellP,cellQ:channel [private]

let P =
 in(c,x:bitstring);
 in(cellP,i:nat);
 let j = sdec(x,k) in
 if j > i
 then
  event Accept(j);
  out(cellP,j)
 else
  out(cellP,i).

let Q =
 in(cellQ,i:nat);
 out(c,senc(I,k));
 out(cellQ,i+1).

process out(cellP,0) | out(cellQ,0) | !P | !Q
```

```
select mess(cellQ[],i_2)/-5000
```

Clauses generated from the process Q

$$mess(cellQ, i) \rightarrow mess(cellQ, i+1)$$

$$\rightarrow mess(cellQ, 0)$$

If $mess(cellQ, i)$ was selected then by resolution:

$$\rightarrow mess(cellQ, 1)$$

$$\rightarrow mess(cellQ, 2)$$

$$\vdots$$

What to do to solve the problem ?

Use a new setting

```
set nounifIgnoreAFewTimes = auto.
```

When solving the query, ProVerif will ignore a « few times » the prevention of resolution.

By default, only one time but it can be parametrized

```
set nounifIgnoreNtimes = 3.
```

The bigger the number, the slower the verification will be

Useful for proofs and finding attacks

Not always enough !

Proof of queries by induction

Simplified Yubikey

```
free k:key [private].
free cellP,cellQ:channel [private]

query i:nat; mess(cellQ,i) ==> is_nat(i).

let P =
 in(c,x:bitstring);
 in(cellP,i:nat);
 let j = sdec(x,k) in
 if j > i
 then
  event Accept(j);
  out(cellP,j)
 else
  out(cellP,i).

let Q =
 in(cellQ,i:nat);
 out(c,senc(I,k));
 out(cellQ,i+1).

process out(cellP,0) | out(cellQ,0) | !P | !Q
```

Even with

```
set nounifIgnoreAFewTimes = auto.
set nounifIgnoreNtimes = 10.
```

With obtain

```
goal reachable: is_not_nat(i_2 + 10) && mess(cellQ[],i_2) ->
mess(cellQ[],i_2 + 10)

…
…

Could not find a trace corresponding to this derivation.
RESULT mess(cellQ[],i_2) ==> is_nat(i_2) cannot be proved.
```

The attacker is untyped !

Proof of queries by induction

Simplified Yubikey

```
free k:key [private].
free cellP,cellQ:channel [private]

query i:nat; mess(cellQ,i) ==> is_nat(i).

let P =
 in(c,x:bitstring);
 in(cellP,i:nat);
 let j = sdec(x,k) in
 if j > i
 then
  event Accept(j);
  out(cellP,j)
 else
  out(cellP,i).

let Q =
 in(cellQ,i:nat);
 out(c,senc(I,k));
 out(cellQ,i+1).

process out(cellP,0) | out(cellQ,0) | !P | !Q
```

```
goal reachable: is_not_nat(i_2 + 10) && mess(cellQ[],i_2) ->
mess(cellQ[],i_2 + 10)
```

The fact `mess(cellQ[],i_2)` occurred strictly before `mess(cellQ[],i_2)` in the trace.

Induction on the size of the trace !

```
query i:nat; mess(cellQ,i) ==> is_nat(i) [induction].
```

Proof of queries by induction

It also works for a group of queries !

Proof by mutual induction

```
query i:nat,…;
 mess(cellQ,i) ==> is_nat(i);

 mess(cellP,i) ==> is_nat(i);

 query_3;

 …

 query_n [induction].
```

As usual it, it may slow down the verification or lead to non-termination

Does not work for injective correspondence

Lemmas, axioms, restrictions

```
restriction phi_1.
…
restriction phi_n.

axiom aphi_1.
…
axiom aphi_m.

lemma lphi_1.

lemma lphi_k.

query attacker(s).
```

Restrictions « restrict » the traces considered in axioms, lemmas and queries.

`query attacker(s).` holds if no trace satisfying `phi_1, …, phi_n` reveals `s`

1 Proverif assumes that the axioms `aphi_1, …, aphi_n` hold.

2 Proverif tries to prove in order the lemmas `lphi_1, …, lphi_k` reusing all axioms and previously proved lemmas

3 Proverif tries to prove the query `query attacker(s).` reusing all axioms and all lemmas.

The precise option under the hood

Option [precise] for inputs, table lookup and predicate testing is coded as an axiom internally .

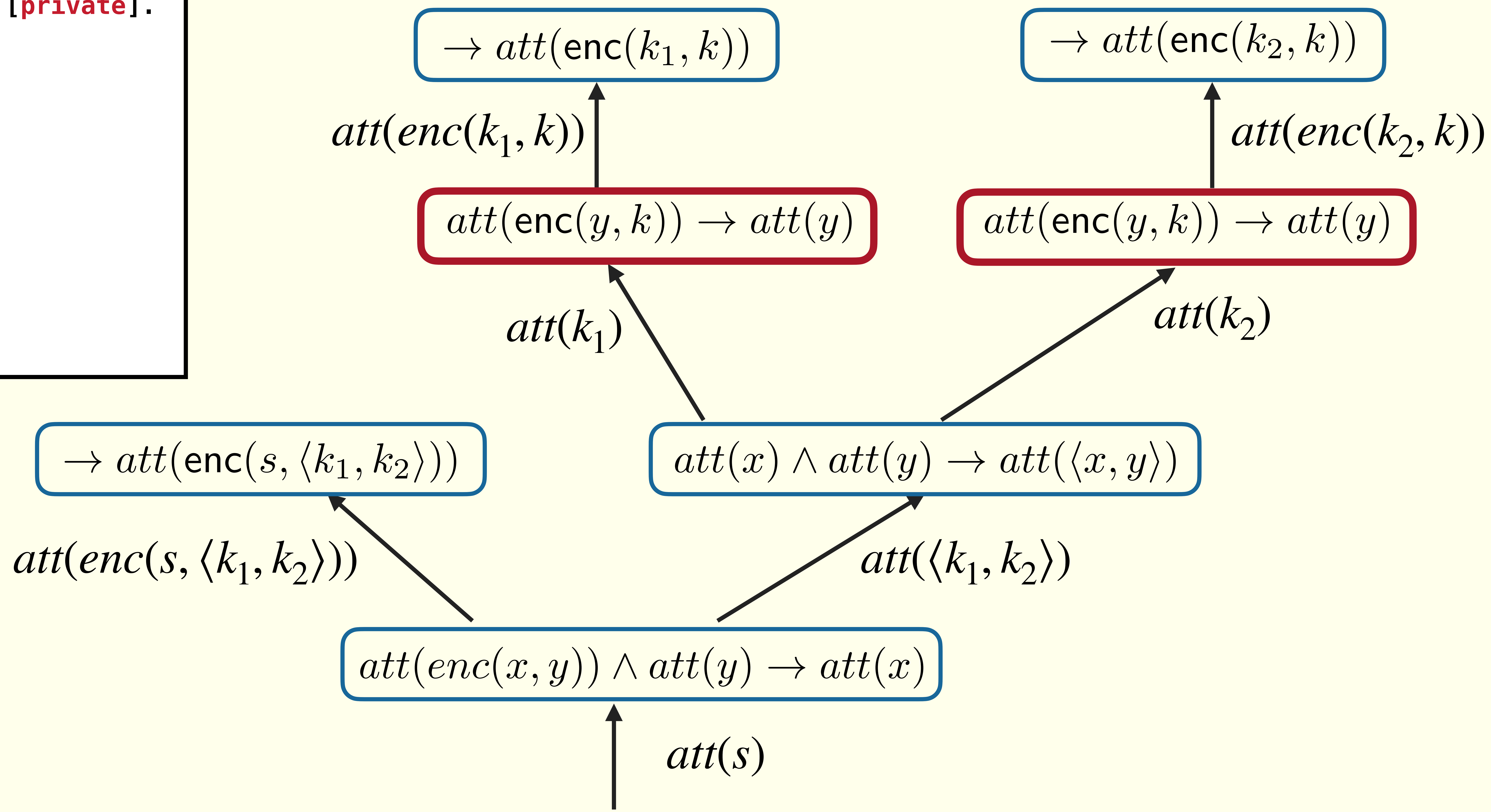
```
free s,k1,k2,k:bitstring [private].

let A =
 out(c,senc(s,(k1,k2)));
 out(c,senc(k1,k));
 out(c,senc(k2,k)).

let B =
 in(c,x) [precise];
 out(c,dec(x,k)).

process A | B
```

Encoded as →

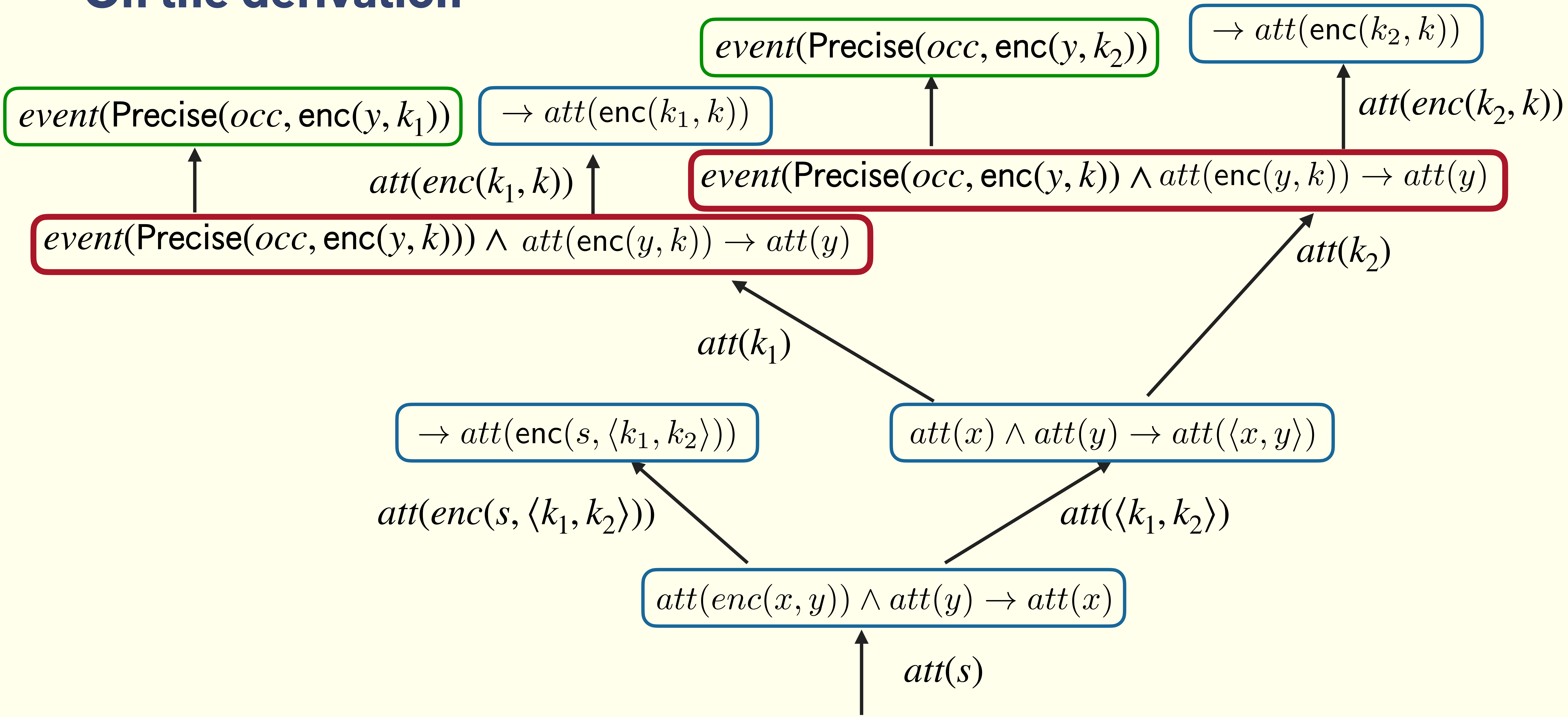
```
type occurrence.

free s,k1,k2,k:bitstring [private].
event Precise(occurrence,bitstring).

axiom occ:occurrence,x1,x2:bitstring;
 event(Precise(occ,x1)) && event(Precise(occ,x2)) ==> x1 = x2.

let A =
 out(c,senc(s,(k1,k2)));
 out(c,senc(k1,k));
 out(c,senc(k2,k)).

let B =
 in(c,x);
 new occ[]:occurrence;
 event Precise(occ,x);
 out(c,dec(x,k)).

process A | B
```

On the derivation

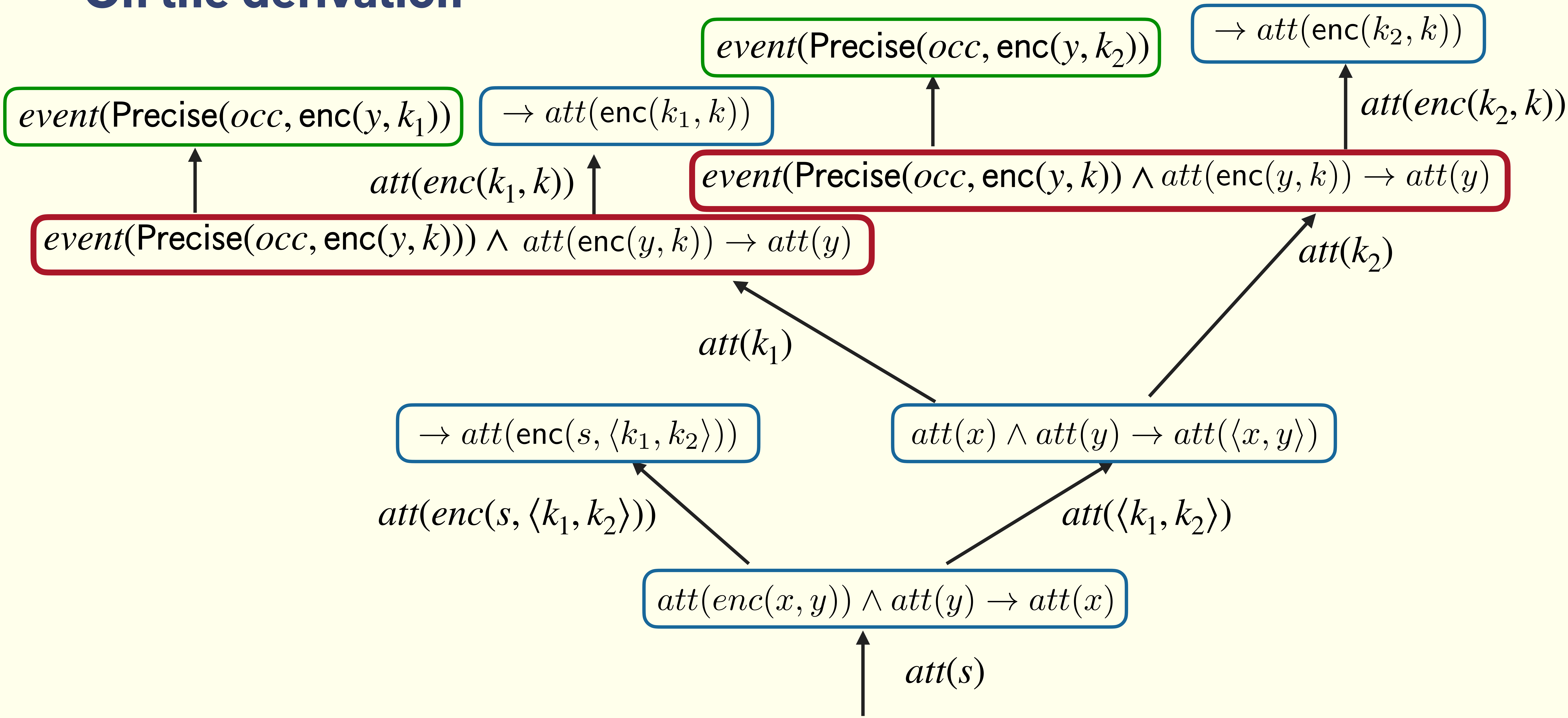
```
free s,k1,k2,k:bitstring [private].

let A =
 out(c,senc(s,(k1,k2)));
 out(c,senc(k1,k));
 out(c,senc(k2,k)).

let B =
 in(c,x);
 out(c,dec(x,k)).

process A | B
```



On the derivation



On the derivation



When to use Lemmas, Axioms and restrictions ?

Restriction

To avoid heavy encoding in the calculus

Ex : To model that a process does not accept twice the same message through multiple session

```
restriction
occ1,occ2:occurrence,x:bitstring;
 event(Unique(occ1,x)) &&
 event(Unique(occ2,x)) ==> occ1 = occ2.

let P =
 in(c,x);
 new occ[]:occurrence;
 event Unique(occ,x);
 …
```

Lemma

Axiom

When you the property can help proving the main query.

Ideally, always use lemma. Use axiom when you can prove by hand (or with another tool) that your property holds … and ProVerif cannot.

Does it work ?

Published protocols

Protocol	Q	O	#	N
PCV Otway-Rees	eq	✗	1	✓
PCV Needham-Schreder	inj	✗	6	✓
			3	ϟ
PCV Denning-Sacco	inj	✗	1	ϟ
JFK	cor	✗	2	ϟ
	inj		2	✓
Arinc823	cor	✗	6	ϟ
Helios-norevote	eq	✗	4	✓
Signal	cor	✗	2	ϟ
TLS12-TLS13-draft18	cor	✗	1	ϟ

Unpublished protocols

Protocol	Q	O	#	N
QBC_4qbits	cor	✗	1	✓
			1	ϟ
Voting-draft	eq	✗	1	✓
LAK-simplified	cor	⏱	1	✓
PACE_v3-sequence	cor	✗	1	✓
			3	ϟ
DP-3T-simpl-draft	cor	⏱	1	✓
			2	ϟ
student1	cor	⏱	2	✓
	inj		1	ϟ
student2	inj	✗	1	✓
student3	cor	⏱	1	ϟ
student4	cor	✗	2	ϟ
student5	cor	⏱	1	ϟ

Outline

A. Additional modeling techniques
   1. Equational Theory vs Rewrite rules
   2. Equivalence properties
   3. Memory cell and locks for stateful protocols
B. Dealing with « cannot be proved »
   1. Adding precision
   2. Trace with assumption
   3. Proof by induction
   4. Lemma
C. Dealing with non-termination

How to determine if ProVerif does not terminate ?

```
Translating the process into Horn clauses...
Completing...
200 rules inserted. Base: 200 rules (97 with conclusion selected). Queue: 679 rules.
400 rules inserted. Base: 400 rules (133 with conclusion selected). Queue: 481 rules.
600 rules inserted. Base: 600 rules (133 with conclusion selected). Queue: 291 rules.
800 rules inserted. Base: 800 rules (133 with conclusion selected). Queue: 135 rules.
1000 rules inserted. Base: 997 rules (157 with conclusion selected). Queue: 184 rules.
1200 rules inserted. Base: 1093 rules (204 with conclusion selected). Queue: 134 rules.
1400 rules inserted. Base: 1253 rules (293 with conclusion selected). Queue: 208 rules.
1600 rules inserted. Base: 1420 rules (352 with conclusion selected). Queue: 281 rules.
1800 rules inserted. Base: 1596 rules (382 with conclusion selected). Queue: 315 rules.
2000 rules inserted. Base: 1790 rules (394 with conclusion selected). Queue: 369 rules.
2200 rules inserted. Base: 1970 rules (400 with conclusion selected). Queue: 387 rules.
2400 rules inserted. Base: 2166 rules (400 with conclusion selected). Queue: 393 rules.
2600 rules inserted. Base: 2323 rules (402 with conclusion selected). Queue: 423 rules.
2800 rules inserted. Base: 2507 rules (402 with conclusion selected). Queue: 447 rules.
3000 rules inserted. Base: 2644 rules (416 with conclusion selected). Queue: 484 rules.
3200 rules inserted. Base: 2790 rules (416 with conclusion selected). Queue: 500 rules.
3400 rules inserted. Base: 2933 rules (443 with conclusion selected). Queue: 547 rules.
3600 rules inserted. Base: 3068 rules (443 with conclusion selected). Queue: 571 rules.
3800 rules inserted. Base: 3209 rules (464 with conclusion selected). Queue: 617 rules.
4000 rules inserted. Base: 3320 rules (484 with conclusion selected). Queue: 715 rules.
4200 rules inserted. Base: 3408 rules (484 with conclusion selected). Queue: 747 rules.
4400 rules inserted. Base: 3529 rules (498 with conclusion selected). Queue: 756 rules.
4600 rules inserted. Base: 3637 rules (530 with conclusion selected). Queue: 804 rules.
4800 rules inserted. Base: 3705 rules (530 with conclusion selected). Queue: 882 rules.
…
…
```

Number of rules generated

Current size of the set of rules

Number of rules left to handle

The first clues

- Size of the queue seems to always increase
- Size of the queue seems to be cyclic
- No rule inserted for ages (can happen with lemmas)
- Termination warnings

Be patient 😊

On TLS 1.3, terminates with 200k rules inserted.

How to determine if ProVerif does not terminate ?

The real way to do it…

```
set verboseRules = true.
```

Display all the rules generated

```
Rule with hypothesis fact 0 selected: mess(cellQ[],i_2)
mess(cellQ[],i_2) -> mess(cellQ[],i_2)
The hypothesis occurs before the conclusion.
1 rules inserted. Base: 1 rules (0 with conclusion selected). Queue: 3 rules.

Rule with hypothesis fact 0 selected: mess(cellQ[],i_2)
is_nat(i_2) && mess(cellQ[],i_2) -> mess(cellQ[],i_2 + 1)
The hypothesis occurs strictly before the conclusion.
2 rules inserted. Base: 2 rules (0 with conclusion selected). Queue: 5 rules.

Rule with conclusion selected:
mess(cellQ[],0)
3 rules inserted. Base: 3 rules (1 with conclusion selected). Queue: 4 rules.

Rule with hypothesis fact 0 selected: attacker(cellQ[])
attacker(cellQ[]) && attacker(i_2) -> mess(cellQ[],i_2)
The 1st, 2nd hypotheses occur before the conclusion.
4 rules inserted. Base: 4 rules (1 with conclusion selected). Queue: 3 rules.

Rule with hypothesis fact 0 selected: mess(cellQ[],i_2)
is_nat(i_2) && mess(cellQ[],i_2) -> mess(cellQ[],i_2 + 2)
The hypothesis occurs strictly before the conclusion.
5 rules inserted. Base: 5 rules (1 with conclusion selected). Queue: 5 rules.

Rule with conclusion selected:
mess(cellQ[],1)
6 rules inserted. Base: 6 rules (2 with conclusion selected). Queue: 4 rules.

Rule with hypothesis fact 0 selected: attacker(cellQ[])
is_nat(i_2) && attacker(cellQ[]) && attacker(i_2) -> mess(cellQ[],i_2 + 1)
The 1st, 2nd hypotheses occur strictly before the conclusion.
7 rules inserted. Base: 7 rules (2 with conclusion selected). Queue: 3 rules.
```

Signs of a cycle

- Size of the term in the conclusion increases
- Number of hypotheses increases

⚠ Very long and painful to read

✔ Best way to find the problem

✔ Best way to understand how to solve it

Not attacker declaration and lemmas

Look if some facts should not be true

```
Rule with hypothesis fact 0 selected: attacker(cellQ[])
attacker(cellQ[]) && attacker(i_2) -> mess(cellQ[],i_2)
The 1st, 2nd hypotheses occur before the conclusion.
4 rules inserted. Base: 4 rules (1 with conclusion selected). Queue: 3 rules.
```

The cell should be private

```
Rule with hypothesis fact 1 selected: attacker(h(i))
is_not_nat(i_2) && event(Accept(i_2)) && attacker(h(i)) -> mess(cellQ[],i_2)
The 1st, 2nd hypotheses occur before the conclusion.
14 rules inserted. Base: 3 rules (2 with conclusion selected). Queue: 3 rules.
```

The content of the cell should be natural numbers

```
not attacker(cellQ).
```

Faster but semantically equivalent to

```
Lemma attacker(cellQ).
```

```
lemma i:nat; mess(cellQ,i) ==> is_nat(i).
```

Playing with the selection function

The fact that will be selected for resolution

```
Rule with hypothesis fact 1 selected: attacker(h(i))
is_not_nat(i_2) && event(Accept(i_2)) && attacker(h(i)) -> mess(cellQ[],i_2)
The 1st, 2nd hypotheses occur before the conclusion.
14 rules inserted. Base: 3 rules (2 with conclusion selected). Queue: 3 rules.
```

The automatic detection of selections to avoid is not perfect

If you think this fact will lead to non-termination, you can tell it to ProVerif

```
noselect i:nat, attacker(h(i)).

noselect i:nat, attacker(h(*i)).
```

`*i` means « any term »

```
select mess(cellQ[],i_2)/-5000
```

Clauses generated from the process Q

$$mess(cellQ, i) \rightarrow mess(cellQ, i + 1)$$

$$mess(cellQ, i_1) \rightarrow mess(cellQ, i_1 + 2)$$
$$mess(cellQ, i_2) \rightarrow mess(cellQ, i_2 + 3)$$
$$\vdots$$

What’s next ?

Development of GSVerif as a frontend and backend

Current: Frontend for stateful protocol

Futur: Backend of ProVerif to suggest user-features to apply

Going beyond diff-equivalence

Modulo XOR / Abelian group

Current: Not accepted by ProVerif

Main concerns: Efficiency and non-termination

Certified ProVerif (very long term)

ProVerif would generate a certificate of a successful verification

Certificate verifier written in F*

Improve memory consumption and efficiency