

ZCash Verif - Formal Analysis of the ZCash Protocol

Vincent Cheval

Inria Paris

vincent.cheval@inria.fr



Lucca Hirschi

Inria Nancy, LORIA

lucca.hirschi@inria.fr

Loria

Steve Kremer

Inria Nancy, LORIA

steve.kremer@inria.fr

informatiques mathématiques
Inria

Vincent Laporte

Inria Nancy, LORIA

vincent.laporte@inria.fr

ZCash protocol

Implementation of the Decentralized Anonymous
Payment scheme Zerocash

Objectives

ProVerif model faithful to the specification

Verification of several security properties

- Balance
- Non-malleability
- Ledger indistinguishability

Following the specification

Zcash Protocol Specification

Version 2020.1.15 [Overwinter+Sapling+Blossom+Heartwood+Canopy]

Daira Hopwood[†]
Sean Bowe[†] – Taylor Hornby[†] – Nathan Wilcox[†]

November 6, 2020

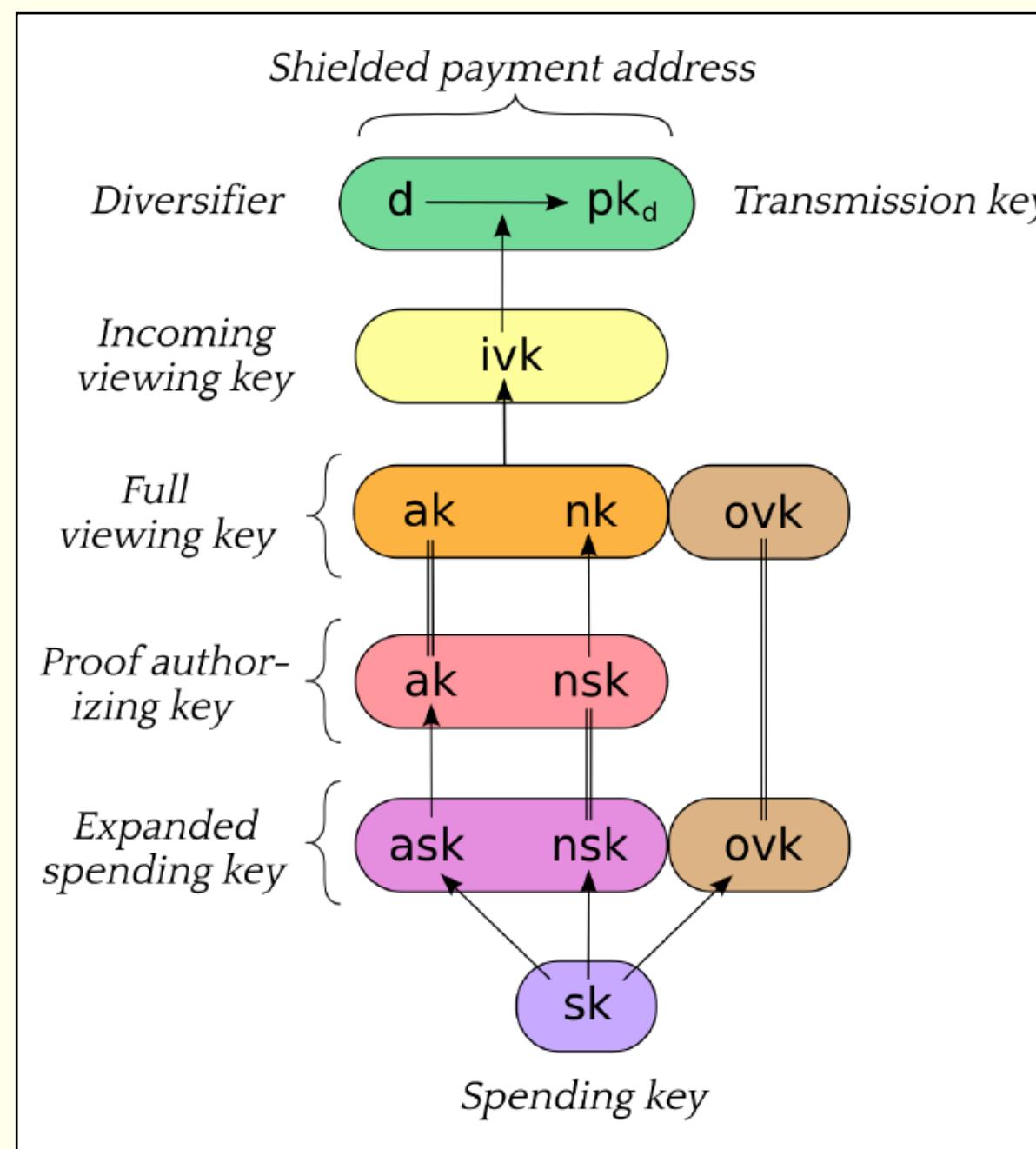


Following the specification

5.4.6.2 Binding Signature	68
5.4.7 Commitment schemes	68
5.4.7.1 Sprout Note Commitments	68
5.4.7.2 Windowed Pedersen commitments	68
5.4.7.3 Homomorphic Pedersen commitments	69
5.4.8 Represented Groups and Pairings	70
5.4.8.1 BN-254	70
5.4.8.2 BLS12-381	71
5.4.8.3 Jubjub	73
5.4.8.4 Hash Extractor for Jubjub	74
5.4.8.5 Group Hash into Jubjub	74
5.4.9 Zero-Knowledge Proving Systems	75
5.4.9.1 BCTV14	75
5.4.9.2 Groth16	76
5.5 Encodings of Note Plaintexts and Memo Fields	77
5.6 Encodings of Addresses and Keys	78
5.6.1 Transparent Addresses	78
5.6.2 Transparent Private Keys	79
5.6.3 Sprout Payment Addresses	79
5.6.4 Sapling Payment Addresses	80
5.6.5 Sprout Incoming Viewing Keys	80
5.6.6 Sapling Incoming Viewing Keys	81
5.6.7 Sapling Full Viewing Keys	81
5.6.8 Sprout Spending Keys	81
5.6.9 Sapling Spending Keys	82
5.7 BCTV14 zk-SNARK Parameters	82
5.8 Groth16 zk-SNARK Parameters	83
5.9 Randomness Beacon	83
6 Network Upgrades	83
7 Consensus Changes from Bitcoin	85
7.1 Transaction Encoding and Consensus	85
7.2 JoinSplit Description Encoding and Consensus	88
7.3 Spend Description Encoding and Consensus	89
7.4 Output Description Encoding and Consensus	89
7.5 Block Header Encoding and Consensus	90
7.6 Proof of Work	92
7.6.1 Equihash	92
7.6.2 Difficulty filter	93
7.6.3 Difficulty adjustment	93
7.6.4 nBits conversion	95
7.6.5 Definition of Work	95
7.7 Calculation of Block Subsidy, Funding Streams, and Founders' Reward	95
7.8 Payment of Founders' Reward	96
5.4.6.2 Binding Signature	68
5.4.7 Commitment schemes	68
5.4.7.1 Sprout Note Commitments	68
5.4.7.2 Windowed Pedersen commitments	68
5.4.7.3 Homomorphic Pedersen commitments	69
5.4.8 Represented Groups and Pairings	70
5.4.8.1 BN-254	70
5.4.8.2 BLS12-381	71
5.4.8.3 Jubjub	73
5.4.8.4 Hash Extractor for Jubjub	74
5.4.8.5 Group Hash into Jubjub	74
5.4.9 Zero-Knowledge Proving Systems	75
5.4.9.1 BCTV14	75
5.4.9.2 Groth16	76
5.5 Encodings of Note Plaintexts and Memo Fields	77
5.6 Encodings of Addresses and Keys	78
5.6.1 Transparent Addresses	78
5.6.2 Transparent Private Keys	79
5.6.3 Sprout Payment Addresses	79
5.6.4 Sapling Payment Addresses	80
5.6.5 Sprout Incoming Viewing Keys	80
5.6.6 Sapling Incoming Viewing Keys	81
5.6.7 Sapling Full Viewing Keys	81
5.6.8 Sprout Spending Keys	81
5.6.9 Sapling Spending Keys	82
5.7 BCTV14 zk-SNARK Parameters	82
5.8 Groth16 zk-SNARK Parameters	83
5.9 Randomness Beacon	83
6 Network Upgrades	83
7 Consensus Changes from Bitcoin	85
7.1 Transaction Encoding and Consensus	85
7.2 JoinSplit Description Encoding and Consensus	88
7.3 Spend Description Encoding and Consensus	89
7.4 Output Description Encoding and Consensus	89
7.5 Block Header Encoding and Consensus	90
7.6 Proof of Work	92
7.6.1 Equihash	92
7.6.2 Difficulty filter	93
7.6.3 Difficulty adjustment	93
7.6.4 nBits conversion	95
7.6.5 Definition of Work	95
7.7 Calculation of Block Subsidy, Funding Streams, and Founders' Reward	95
7.8 Payment of Founders' Reward	96
7.9 Payment of Funding Streams	98
7.9.1 ZIP 214 Funding Streams	99
7.10 Changes to the Script System	99
7.11 Bitcoin Improvement Proposals	100
8 Differences from the Zerocash paper	100
8.1 Transaction Structure	100
8.2 Memo Fields	100
8.3 Unification of Mints and Pours	100
8.4 Faerie Gold attack and fix	101
8.5 Internal hash collision attack and fix	102
8.6 Changes to PRF inputs and truncation	103
8.7 In-band secret distribution	104
8.8 Omission in Zerocash security proof	105
8.9 Miscellaneous	105
9 Acknowledgements	106
10 Change History	107
11 References	126
Appendices	137
A Circuit Design	137
A.1 Quadratic Constraint Programs	137
A.2 Elliptic curve background	137
A.3 Circuit Components	138
A.3.1 Operations on individual bits	138
A.3.1.1 Boolean constraints	138
A.3.1.2 Conditional equality	139
A.3.1.3 Selection constraints	139
A.3.1.4 Nonzero constraints	139
A.3.1.5 Exclusive-or constraints	139
A.3.2 Operations on multiple bits	139
A.3.2.1 [Un]packing modulo r_S	139
A.3.2.2 Range check	140
A.3.3 Elliptic curve operations	142
A.3.3.1 Checking that Affine-ctEdwards coordinates are on the curve	142
A.3.3.2 ctEdwards [de]compression and validation	142
A.3.3.3 ctEdwards \leftrightarrow Montgomery conversion	142
A.3.3.4 Affine-Montgomery arithmetic	143
A.3.3.5 Affine-ctEdwards arithmetic	144
A.3.3.6 Affine-ctEdwards nonsmall-order check	145
A.3.3.7 Fixed-base Affine-ctEdwards scalar multiplication	145
A.3.3.8 Variable-base Affine-ctEdwards scalar multiplication	146
A.3.3.9 Pedersen hash	147

Following the specification

Keys hierarchy



```
(* Expanded spending key *)
letfun ask(sk:t_sk) = ToScalar(PRF_expand(sk, zero)).
letfun nsk(sk:t_sk) = ToScalar(PRF_expand(sk, one)).
(* We do not currently consider "weak" keys like 0 resulting from some PRF calls.
   We may want to add the possibility of getting weak keys when honest agents generating them do not
   explicitly do non zero checks. *)

(* Proof authorizing key *)
letfun ak(ask:t_scalar) = Sign_DerivePublic(Sign_prv_OF_scalar(ask)).

(* Full viewing key *)
letfun nk(nsk:t_scalar) = J_OF_KA_pub_key(KA_DerivePublic(KA_prv_key_OF_scalar(nsk), basePoint)).
(* Note that we model scalar multiplication in the group J using KA's group since we concluded that
   the implementation of KA is based on the group J as well. *)

(* Incoming viewing key *)
letfun ivk(ak:t_Sign_pub,nk:t_J) = CRH_ivk(J_OF_Sign_pub(ak), nk).

(* To generate a diversified payment address *)
letfun pkd_(d:t_d,ivk:t_ivk) = KA_DerivePublic(KA_prv_key_OF_scalar(scalar_OF_ivk(ivk)),
KA_pub_key_OF_Jstar(Diversify_hash(d))). 

letfun d0Fsk(d:t_d, sk:t_sk) = pkd_(d,ivk(ak(ask(sk)),nk(nsk(sk)))).

letfun div_pay_addr(ivk:t_ivk) = new d:t_d; (d,pkd_(d,ivk)). 

(* Outgoing viewing key *)
letfun ovk(sk:t_sk) = truncate_ovk(PRF_expand(sk, two)).
```

Ledger

Limitation

Global consensus on the ledger

Limitation

Tree structure of ledger not represented

Entries of the commitment trees in a table (set)

```
(* The memory cell [size_tree] will contain the current number of elements
contained in the tree. *)
free size_tree:channel [private].  
  
(* The anchor is modeled by applying the private function on the number of elements
in the tree. Note that the modeling only works with a unique, global treestate.
*)
fun gen_anchor(nat):t_J [private]. (* Root of Merkle Tree *)
fun gen_pos(nat):random.          (* Position in the Merkle tree *)
fun gen_path(nat):random.         (* Path in the Merkle tree *)
  
table commitment_trees(nat,identity,random,random,t_J).
```

Verify transaction (Insert)

```
in(size_tree,cur_size:nat);
in(0_insert,tr:bitstring);    (* The transaction *)
...
Verify spend descriptions
...
Verify output descriptions with two note commitments:
cm_o1, cm_o2
...
Verify Balance
...
  

let path_o1 = gen_path(cur_size+1) in
let pos_o1 = gen_pos(cur_size+1) in
let path_o2 = gen_path(cur_size+2) in
let pos_o2 = gen_pos(cur_size+2) in
insert commitment_trees(cur_size+1,id, path_o1, pos_o1, cm_o1);
insert commitment_trees(cur_size+2,id, path_o2, pos_o2, cm_o2);
out(size_tree,cur_size+2);
```

Limitation

Two inputs, Two outputs

Ledger

Entries of the nullifier sets by events

(Test « not in table » badly handled by ProVerif)

```
event UniqueNullifier(bits_256,identity,stamp).
```

```
(* Excludes traces that contain two UniqueNullifier with the same nullifier
[nf] *)
restriction nf:bits_256, st1, st2:stamp, id:identity;
  event(UniqueNullifier(nf,id,st1)) &&
  event(UniqueNullifier(nf,id,st2)) ==>
    st1 = st2.
```

Verify transaction (Insert)

```
in(size_tree,cur_size:nat);
in(0_insert,tr:bitstring); (* The transaction *)
...
Verify spend descriptions with two nullifiers nf_i1, nf_i2
new t1:stamp;
event UniqueNullifier(nf_i1,id,t1);
new t2:stamp;
event UniqueNullifier(nf_i2,id,t2);

...
Verify output descriptions with two note commitments:
cm_o1, cm_o2
...
Verify Balance
...
let path_o1 = gen_path(cur_size+1) in
let pos_o1 = gen_pos(cur_size+1) in
let path_o2 = gen_path(cur_size+2) in
let pos_o2 = gen_pos(cur_size+2) in
insert commitment_trees(cur_size+1,id, path_o1, pos_o1, cm_o1);
insert commitment_trees(cur_size+2,id, path_o2, pos_o2, cm_o2);

out(size_tree,cur_size+2);
```

Modeling cryptographic primitives

Hash functions and PRF



Perfect one-way function

TOO STRONG

```
fun h(bitstring):bitstring.
```

CRH^{ivk} , Mixing Pedersen Hash

Collision resistant, not second-image resistant, not unlinkable

DiversifyHash

Unlinkable, not collision resistant

SigHash

Unlinkable, not second-image resistant

Modeling cryptographic primitives

Adds equations and destructors to model primitives weaknesses

Not collision resistant

```
fun coll_a(bitstring):t_input.  
fun coll_b(bitstring):t_input.  
  
equation forall x:bitstring; h(coll_a(x)) = h(coll_b(x)).
```

Linkable

```
fun isLinked1(t_output,t_output) : bool  
reduc forall x:t_input1, y:t_input2, z:t_input2; isLinked1(h(x,y),h(x,z)) = true.  
  
fun isLinked2(t_output,t_output) : bool  
reduc forall x:t_input2, y:t_input1, z:t_input1; isLinked2(h(y,x),h(z,x)) = true.
```

4 levels of security

Basic

Weak (Linkable, etc)

Collapse (to a single value)

Inversible

Limitation

Best effort

Modeling cryptographic primitives

Signature weaknesses

Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. Seems legit: Automated analysis of subtle attacks on protocols that use signatures.

No Conservative Exclusive Ownership

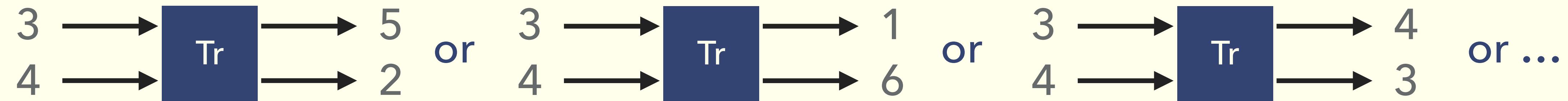
```
fun CE0gen(t_sig): t_prv.  
  
fun validate(t_pub,t_msg,t_sig) : bool  
reduc  
  forall x:t_prv,y:t_msg,r:random; validate(derive_public(x),y,i_sign(x,y,r)) = true  
otherwise  
  forall x:t_prv,y:t_msg,r:random; validate(derive_public(CE0gen(i_sign(x,y,r))),y,i_sign(x,y,r)) = true.
```

Destructive Exclusive Ownership, Collision signatures...

Modeling cryptographic primitives

Binding signatures and balance

Ideally



Limitation

No arithmetic

In our model: only swap possible



Oracle processes

Followed the « oracle-style » from cryptographic games

Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza.
Zerocash: Decentralized Anonymous Payments from Bitcoin.

Create address

Generates new honest agents with their spending key (sk) and unbounded number of payment addresses (d,pkd).

Mint

Bootstrap some coins and adds them to ledger. Only honest agent can mint.

Pour

Enforce honest agents to compute a transaction spending and creating coins.

Insert

Receives a transaction from attacker and adds it to ledger if valid.

Oracle processes

Create address

```
channel 0_ca.

let CreateAddress =
  (* Honest Agent Creations *)
  new sk:t_sk;
  new sk_PubId:pubID;
  (* Public pointer to a secret key (notably used by the
   adversary) for instructing honest transactions. *)
  out(0_ca,sk_PubId);
  insert spendingKeys(sk_PubId, sk, IDstorage);

  ! (* Generate set of (d,pkd) revealed to the attacker *)
  new d:t_d;
  let pkd: t_KA_pub_key = d0Fsk(d, sk) in
  new pkd_PubId:pubID;

  out(0_ca,(pkd_PubId,d,pkd));
  insert paymentAddresses(pkd_PubId, pkd, d, sk, IDstorage)
.
```

Insert

```
channel 0_insert.

let Insert =
  in(0_insert, tr:bitstring);

  (* Verify the unique transaction *)
  in(size_tree,cur_size:nat);

  if verifyTransaction22(cur_size,IDstorage,tr) then
    (* This will add the created note commitments and spend
     nullifiers to the treestate. *)
    out(size_tree,cur_size+2)
  else
    out(size_tree,cur_size)
.
```

Security properties

- Non-malleability
- Balance
- ~~Ledger indistinguishability~~

Non-malleability

The attacker cannot produce a transaction tr sharing a nullifier with a different honest transaction tr' with tr valid except for the shared nullifier already in the nullifier set.

Non malleability

Obtained from the attacker

Obtained from Pour oracle

```
let checkMalleability(id:identity, tr:bitstring, tr':bitstring) =  
  
...  
  
if tr <> tr' then  
  
  (** Check that tr and tr' spend the same nullifier (for the first coin they spend). *)  
  if nf_i1 = nf_i1' then  
  
    (** Verify the spending/output descriptions *)  
    let verifyZK_i1: bool = ... in  
    let verifySign_i1: bool = ... in  
  
    ...  
    let verifyZK_o2: bool = ... in  
  
    (** Verify balance *)  
    let verifyBalance: bool = isBalance22(bindSig,sig_hash,cv_i1,cv_i2,cv_o1,cv_o2) in  
  
    if verifyZK_i1 && verifySign_i1 && verifyZK_i2 && verifySign_i2 && verifyZK_o1 && verifyZK_o2 && verifyBalance  
    then  
      if nf_i2 = nf_i2'  
      then event transactionIsMalleable  
      else  
        (** Verify that the second nullifier is unspent if different *)  
        if check_and_insert_nullifier(id,nf_i2) then  
          event transactionIsMalleable
```

```
query event(transactionIsMalleable) ==> false.
```

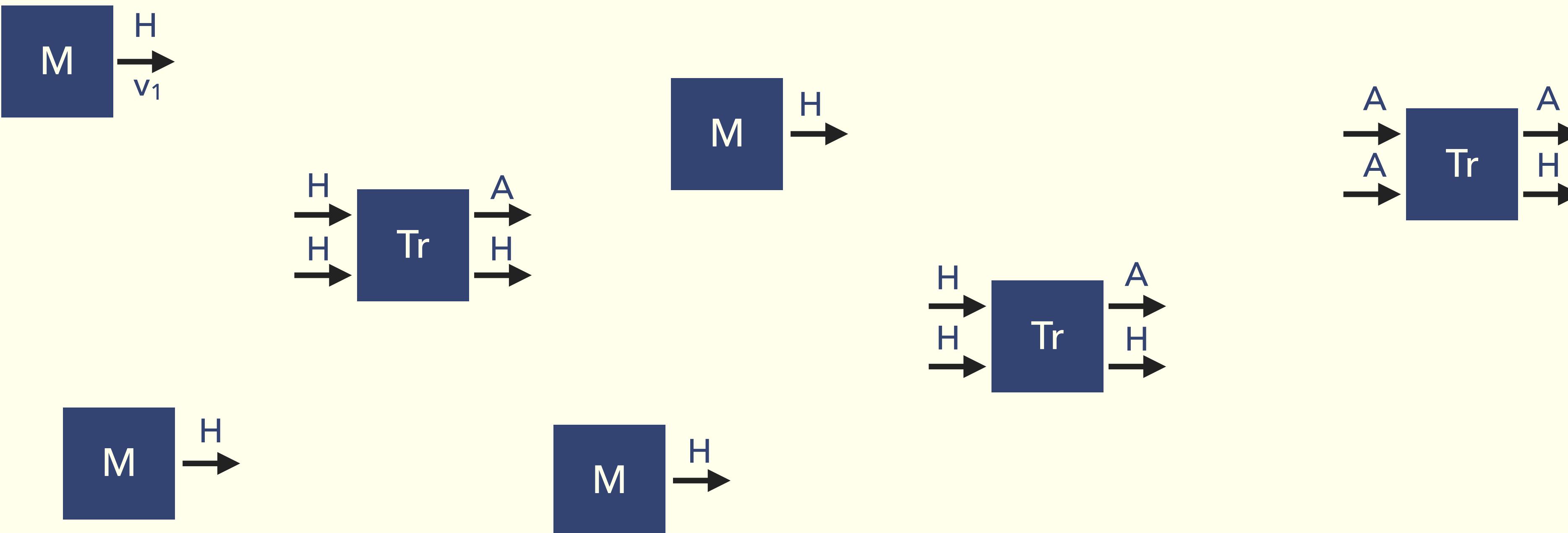
Balance

Ideally: $v_{\text{unspentA}} + v_{A \rightarrow H} \leq v_{H \rightarrow A}$

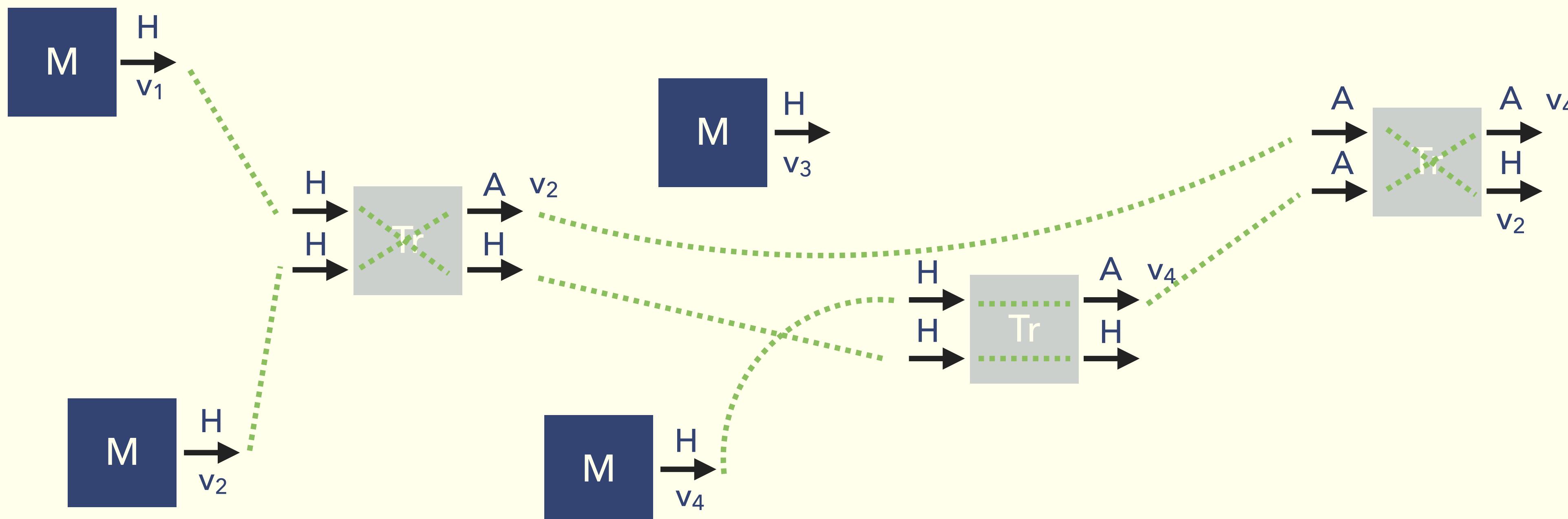
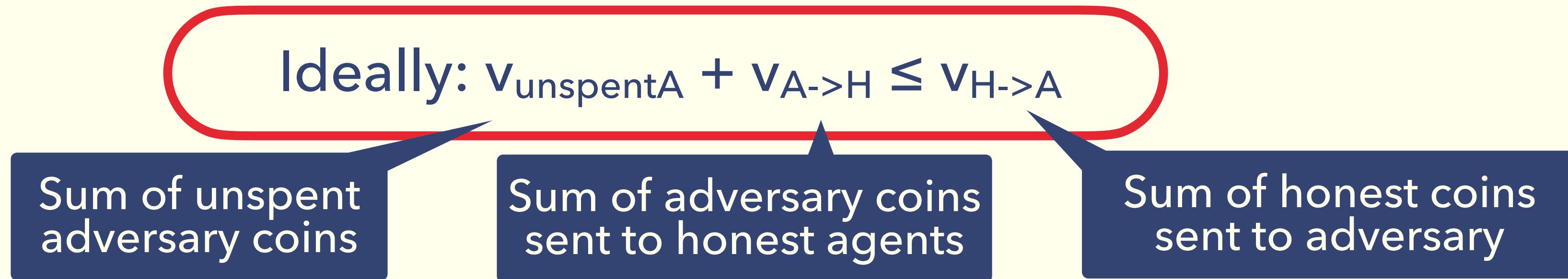
Sum of unspent
adversary coins

Sum of adversary coins
sent to honest agents

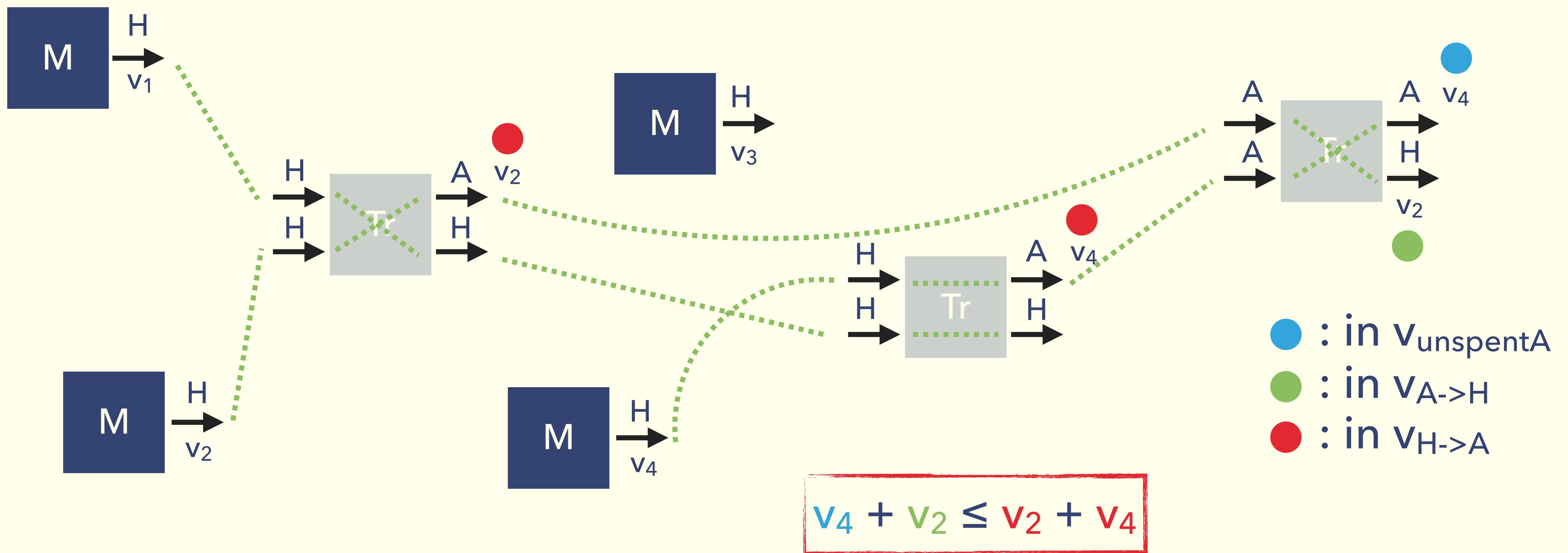
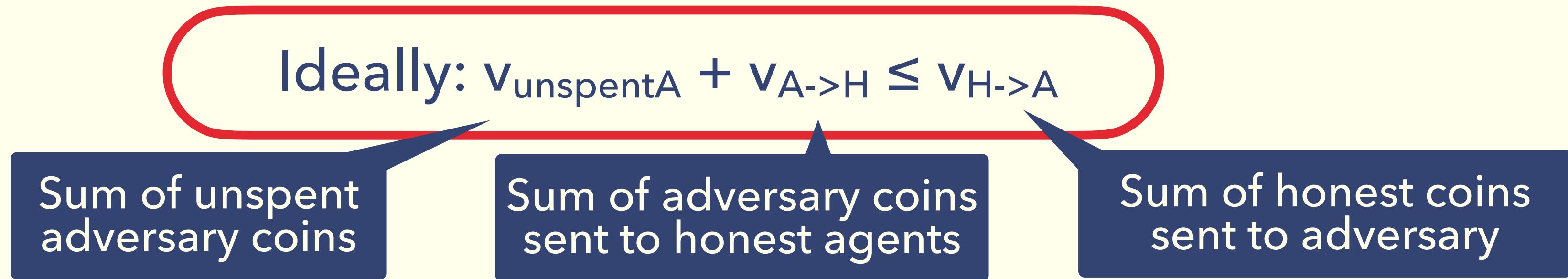
Sum of honest coins
sent to adversary



Balance



Balance



Results

	Balance	Malleability
BASIC	✓	✓
WEAK	✓	✓
INV	✓	✓
COLLAPSE	✗	✓
BROKEN_PRF	✓	✗

✗ : ProVerif could not prove (does not necessarily mean attack)

✓ : proved

What's next ?

Ledger indistinguishability

Weaken ledger consensus hypothesis

Allow more than value swapping

Allow more than 2 in / 2 out in transactions