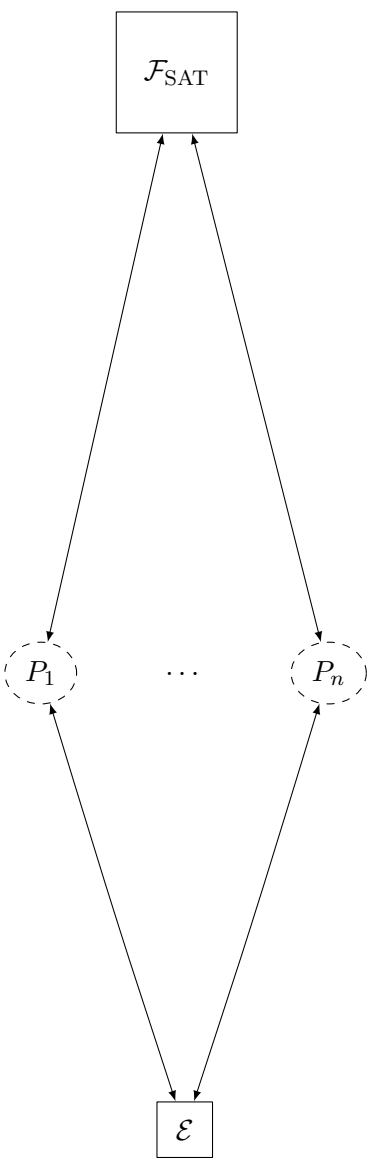
# What is Trust

Orfeas Stefanos Thyfronitis Litos

University of Edinburgh
o.thyfronitis@ed.ac.uk

**Abstract.** We will try to define all the abstract properties that we would like "Trust" to have.

## 1 High-level idea



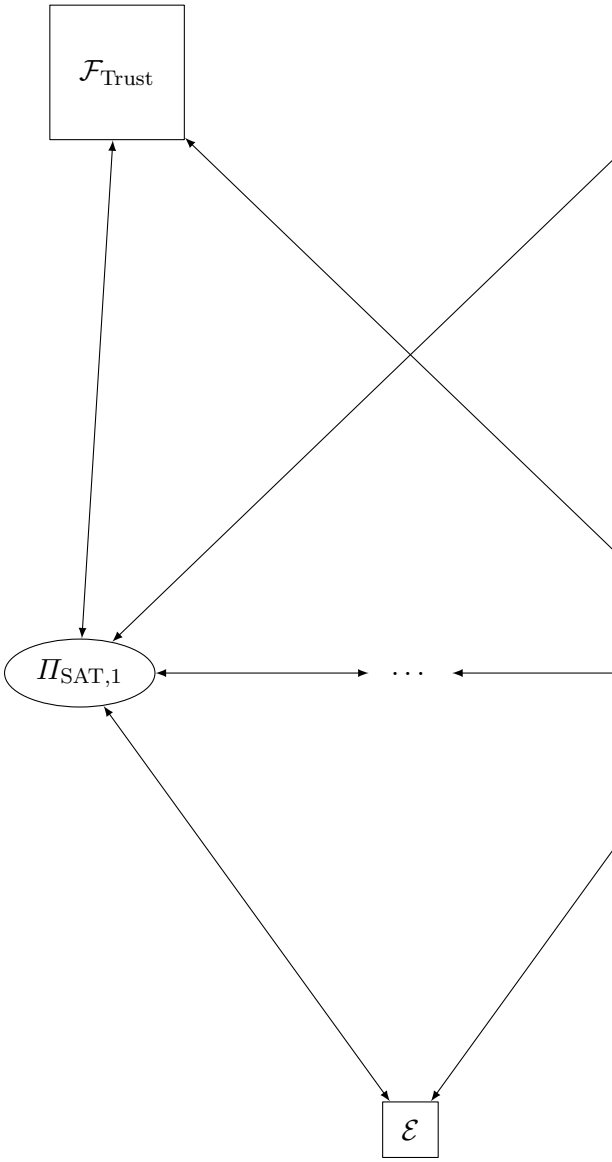
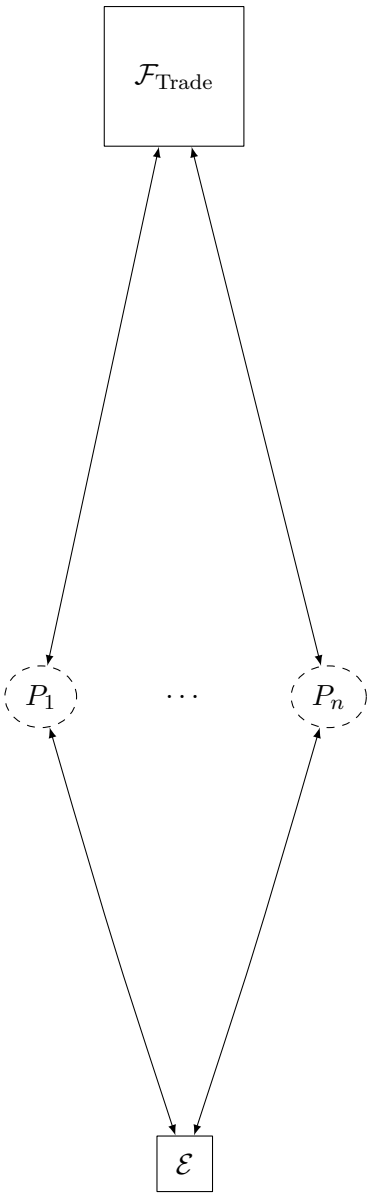




**Fig. 1:** (Almost) all functionalities



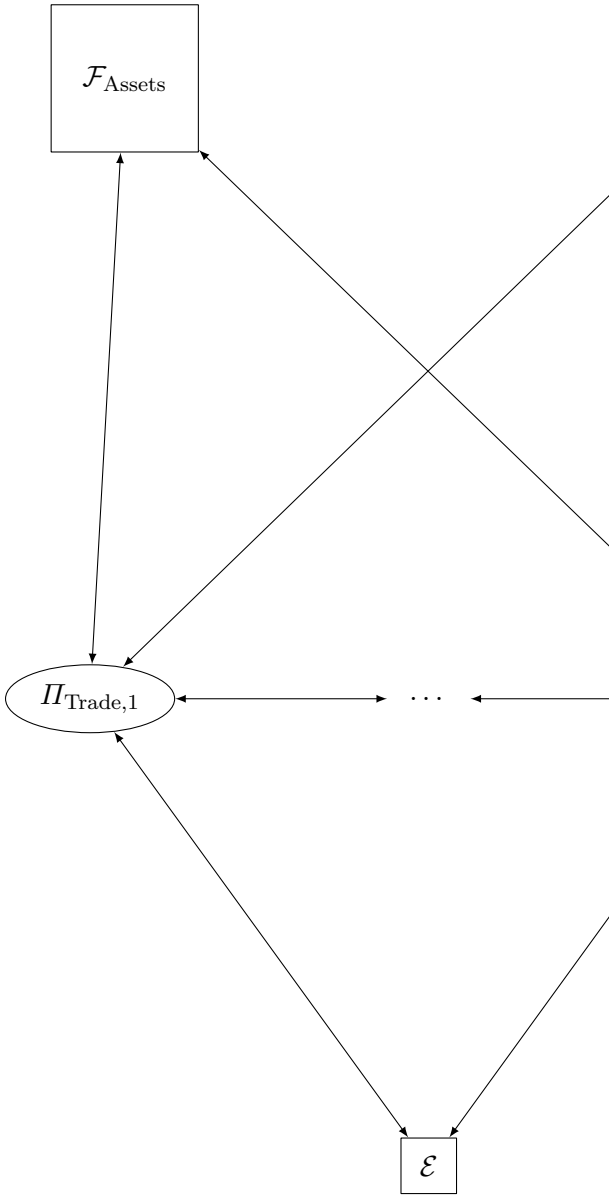




**Fig. 2:** Trade functionality and protocol

## 2 Utility Function Properties

Before *Alice* (an ITM that executes $\Pi_{\text{SAT}}$) can take any action, she must be assigned a utility function $U_{Alice} : Time \times Money \times multiset\,(Asset) \to \mathbb{R}$, where $Money = Time = \mathbb{N}$. The utility function is strictly increasing with respect to $Money$ and to the quantity of any single $Asset$ or combination of $Assets$.

$\mathcal{E}$ can decide to conclude the game at any time $t \in Time$; *Alice*'s utility will be then determined by

$$U_{Alice}\left(t, \mathcal{G}_{\text{Ledger}}.\texttt{money}(Alice), \mathcal{G}_{\text{Assets}}.\texttt{assets}(Alice)\right) \quad .$$

## 3 Protocol

In this section we will describe the real protocol executed by the players in the absence of the $\mathcal{F}_{\text{SAT}}$ ideal functionality. The description of this protocol does not need any utility function; all the "important" decisions are taken by the environment.

Consider an environment $\mathcal{E}$, and adversary $\mathcal{A}$ and $n$ players executing copies of the same protocol $\Pi_1, \ldots, \Pi_n$. $\mathcal{E}$ can send the following messages to $\Pi_i$:

1. Manage player desires
   (a) Satisfy $d \in \mathcal{D}$ through a player in $L \subseteq [n]$
   (b) Abort attempt to satisfy $d \in \mathcal{D}$
2. Manage offered desires satisfaction
   (a) Gain the ability to satisfy $d \in \mathcal{D}$ for players in $L \subseteq [n]$ for a price $x \in \mathbb{N}$
   (b) Lose the ability to satisfy $d \in \mathcal{D}$ for players in $L \subseteq [n]$ for a price $x \in \mathbb{N}$
3. Satisfy another player's desire
   (a) Satisfy player's $i \in [n]$ desire $d \in \mathcal{D}$ with the corresponding satisfaction string $s$
   (b) Satisfy player's $i \in [n]$ desire $d \in \mathcal{D}$ with the satisfaction string $s'$ (normally suitable for satisfying $d' \neq d, d' \in \mathcal{D}$)
   (c) Ignore player's $i \in [n]$ desire $d \in \mathcal{D}$
4. Manage direct trusts
   (a) Increase direct trust to player $i \in [n]$ by $x \in \mathbb{N}$
   (b) Decrease direct trust to player $i \in [n]$ by $x \in \mathbb{N}$

(c) Steal direct trust $x \in \mathbb{N}$ from player $i \in [n]$

Some of these messages (e.g. 1b) are meaningful only when some other messages have been delivered previously (e.g. 1a). $\mathcal{E}$ may send such messages even when they are not meaningful; the protocol should take care to reject/ignore such messages.

Let $i \in [n]$. $\Pi_i$ can send the following messages to $\mathcal{E}$:

1. No player in $L$ can satisfy my desire $d \in \mathcal{D}$
2. Desire $d \in \mathcal{D}$ made available for satisfaction amongst $L \subseteq [n]$ for price $x \in \mathbb{N}$
3. Desire $d \in \mathcal{D}$ made unavailable for satisfaction amongst $L \subseteq [n]$ for price $x \in \mathbb{N}$
4. Payment $x \in \mathbb{N}$ has been sent to player $j \in [n]$ for the satisfaction of desire $d \in \mathcal{D}$
5. Correct Payment $x \in \mathbb{N}$ from player $j \in [n]$ for the satisfaction of desire $d \in \mathcal{D}$ has been received
6. Wrong Payment $x \in \mathbb{N}$ from player $j \in [n]$ for the satisfaction of desire $d \in \mathcal{D}$ has been received
7. Player $j \in [n]$ has satisfied my desire $d \in \mathcal{D}$ with satisfaction string $s$
8. Player $j \in [n]$ has partially satisfied my desire $d \in \mathcal{D}$ with satisfaction string $s'$ (normally suitable for satisfying $d' \neq d, d' \in \mathcal{D}$)
9. Player $j \in [n]$ has ignored my desire $d \in \mathcal{D}$
10. Direct trust to player $i \in [n]$ increased by $x \in \mathbb{N}$
11. Direct trust to player $i \in [n]$ decreased by $x \in \mathbb{N}$
12. Stole $x \in \mathbb{N}$ from player's $i \in [n]$ direct trust

$\Pi_i$ should only send these messages when $\mathcal{E}$ is expecting them.

Let $i, j \in [n]$ The messages that can be sent between $\Pi_i$ and $\Pi_j$ are the following:

1. Can you satisfy $d \in \mathcal{D}$?
2. I can satisfy $d \in \mathcal{D}$ for a price $x \in \mathbb{N}$
3. I cannot satisfy $d \in \mathcal{D}$
4. Payment of $x \in \mathbb{N}$ for satisfaction of $d \in \mathcal{D}$ sent
5. Satisfaction string $s'$, response to payment of $x \in \mathbb{N}$ for $d \in \mathcal{D}$

$\Pi_i$ is supposed to send 4 when it has already paid through $\mathcal{F}_{\text{Ledger}}$. Similarly, it is supposed to send 5 when it has verified that the other party has sent the corresponding payment on $\mathcal{F}_{\text{Ledger}}$.

Going in more detail, the actual protocol is as follows:

$\Pi_{\text{SAT}}$

```
Initalization:
  util = ⊥

Upon receiving (utility, U) from ℰ:
  util = U

Upon receiving message (satisfy, d, L) from ℰ at time t:
  If util == ⊥:
    send message (utilityNotSet) to ℰ
    go to Idle state
  aux = L
  m = 𝒢_Ledger.money(Alice)
  a = 𝒢_Assets.assets(Alice)
  While (aux ≠ ∅):
    send message (chooseBestSeller, d, aux) to ℱ_Trust
    wait for response1 from ℱ_Trust
    If response1 == (bestSeller, d, L, Bob):
      send message (canYouSatisfy, d) to Bob
      wait for response2 from Bob
      If response2 == (IcanSatisfy, d, x, s):
        If util(t+1, m - x, a ∪{s}) > util(t, m, a) and s ∈
            d
          send message (trade, x, s, Bob) to ℱ_Trade
          wait for response3 from ℱ_Trade
            If response3 == (traded, x, s, Bob):
              send message (satisfied, d, x, s, Bob) to ℱ_Trust
              # maybe send only utility difference to ℱ_Trust
              send message (satisfied, d, L) to ℰ
              go to Idle state
            Else If response3 == (cheated, x, s, Bob):
              send message (cheated, d, x, s, Bob) to ℱ_Trust
              send message (cheated, d, L) to ℰ
              go to Idle state
    Else: # if response1 == ⊥
      send message (unsatisfied, d, L) to ℰ
      go to Idle state
    aux = aux \{Bob} # only when response2 is not good
  send message (unsatisfied, d, L) to ℰ
```

```
Upon receiving message (obtain, s) from ℰ:
  send message (obtain, s) to ℱ_Trade
  wait for response from ℱ_Trade
  If response == (obtained, s):
    send message (obtained, s) to ℰ
  Else:
    send message (notObtained, s) to ℰ

Upon receiving message (lose, s) from ℰ:
  send message (lose, s) to ℱ_Trade
  wait for response from ℱ_Trade
  If response == (lost, s):
    send message (lost, s) to ℰ
  Else:
    send message (notLost, s) to ℰ

Upon receiving message (canYouSatisfy, d) from Alice at
    time t:
  If util == ⊥:
    ignore request, go to Idle state
  m = 𝒢_Ledger.money(Alice)
  a = 𝒢_Assets.assets(Alice)
  If ∃s ∈ a ∩ d, x ∈ Money :
    (util(t + 1, m + x, a \ {s}) > util(t, m, a) or
    (util(t + 1, m + x, a) > util(t, m, a) and I do not
        care for the bad rep))
      send message (IcanSatisfy, d, x, s) to Alice
  Else:
    send message (IcannotSatisfy, d) to Alice

Upon receiving message (willWeCheat, x, s, Alice) from
    ℱ_Trade:
  If util == ⊥: # (Unreachable since we've already engaged)
    ignore request, go to Idle state
  If cheating is a bad idea:
    send message (doNotCheat, x, s, Alice) to ℱ_Trade
  Else:
    send message (cheat, x, s, Alice) to ℱ_Trade
```

## 4 Desire Satisfaction Ideal Functionality

Following the UC paradigm, in this section we define the ideal functionality for desire satisfaction, $\mathcal{F}_{\text{SAT}}$. In this setting, all the desires that are generated by the environment and are input to the players are immediately forwarded to $\mathcal{F}_{\text{SAT}}$; the functionality decides which desires to satisfy. Since the players are dummy and all desires are satisfied by the functionality, no trust semantics amongst the players are necessary.

Nevertheless, given that all desires have a minimum cost, the cost semantics are still necessary, as we show with the following example: Consider a set of desires $D$ with more elements than the total number of tokens all players have. $D$ could never be satisfied by the players because of the high total cost, but a $\mathcal{F}_{\text{SAT}}$ with no consideration for cost could in principle satisfy all desires in $D$.

The functionality can calculate the properties and functions defined in **??**, **??** and **??** for all inputs at any moment in time.

Without knowledge of the utilities the environment is going to give to each satisfied desire, the functionality may fail spectacularly. So knowledge of the utility of each desire, or at least some function of the utility given the desires is needed. We can assume that $\mathcal{F}_{\text{SAT}}$ knows $U$ or an approximation of it.

Going into more detail, $\mathcal{F}_{\text{SAT}}$ is a stateful process that acts as a market and as a bank for the players. The market does not offer a particular product for the same price to all users; For some users it may be cheaper than for others, reflecting the fact that some players can realize some desires more efficiently than others.

$\mathcal{F}_{\text{SAT}}$ stores a number for each player that represents the amount of tokens this player has and a table with the price of each desire for each player. It provides the functions $cost\,(u, d)$ which returns the cost of the desire $d$ for player $u$ with no side effects, $sat\,(u, d)$ that returns the string that satisfies the desire $d$ to $u$ and reduces the amount of the tokens belonging to $u$ by $cost\,(u, d)$. There exists also the function $transfer\,(u_1, u_2, t)$ which reduces the amount of tokens $u_1$ has by $t$ and increases the tokens of $u_2$ by $t$, given that initially the tokens belonging to $u_1$ were equal or more than $t$. This function is private to the functionality, thus can be used only internally.

$\mathcal{F}_{\text{Trade}}$

```
1 Initialisation:
2   ∀Alice ∈ 𝒫,
3     assets(Alice) = ⊥
```

```
4
5  Upon receiving (trade, ours, theirs, Bob) from Alice:
6    If not isAvailable(ours, Alice):
7      send message (youDontHave, ours) to Alice
8      go to Idle state
9    If transfer(ours, Alice, Bob) == True:
10     send message (willWeCheat, ours, theirs, Alice) to Bob
11     wait for response from Bob
12     If (response == (complete, x, s, Alice) and
13       not isAvailable(s, Bob)) or
14       response ≠ (complete, x, s, Alice):
15         send message (youDontHave, s) to Bob
16         send message (cheated, ours, theirs, Bob) to Alice
17         go to Idle state
18     Else If (transfer, theirs, Bob, Alice) == True:
19       send message (traded, ours, theirs, Bob) to Alice
20   Else # failed to give (Unreachable for a good F_Ledger)
21     send message (failed, ours, theirs, Bob) to Alice
22
23 isAvailable(object, player):
24   If object is money:
25     send (doIhaveBalance, object) to F_Ledger as player
26     wait for response from F_Ledger
27     return response
28   Else: # object is asset
29     If object ∈ assets(player):
30       return True
31     Else:
32       return False
33
34 transfer(object, sender, receiver):
35   If isAvailable(object):
36     If object is money:
37       send (pay, object, receiver) to F_Ledger as sender
38       wait for response from F_Ledger
39       Upon receiving (paymentDone, object, receiver):
40         return True
41     Else: # object is asset
42       assets(sender) = assets(sender) \ {object}
43       assets(receiver) = assets(receiver) ∪ {object}
```

```
      return True
  return False

Upon receiving (obtain, s) from Alice:
  assets(Alice) = assets(Alice)∪{s}
  send message (obtained, s) to Alice

Upon receiving (lose, s) from Alice:
  assets(Alice) = assets(Alice)\{s}
  send message (lost, s) to Alice
```

$\mathcal{F}_{\text{Assets}}$

```
Initialisation:
  ∀Alice ∈ P,
    assets(Alice) = ⊥

Upon receiving (add, asset) from Alice:
  assets(Alice) = assets(Alice)∪{asset}
  send (added, asset) to Alice

Upon receiving (remove, asset) from Alice:
  If asset ∈ assets(Alice):
    assets(Alice) = assets(Alice)\{asset}
    send (removed, asset) to Alice
  Else:
    send (unableToRemove, asset) to Alice

Upon receiving (howManyDoIhave, asset) from Alice:
  send (youHave, assets(Alice).count(asset)) to Alice

Upon receiving (transfer, asset, Bob) from Alice:
  If asset ∈ assets(Alice):
    assets(Alice) = assets(Alice)\{asset}
    assets(Alice) = assets(Bob)∪{asset}
    send (transferred, asset, Bob) to Alice
  Else:
    send (unableToTransfer, asset, Bob) to Alice
```

$\Pi_{\text{Trade}}$

```
Upon receiving (trade, ours, theirs, Bob) from E:
  Send (letsTrade, ours, theirs) to Bob
```

```
  If transfer(ours, Bob) == True:
    Send (transferred, ours, Bob) to Bob and ℰ
    Wait for response from Bob
    If response == (transferred, theirs, Bob):
      send message (traded, ours, theirs, Bob) to ℰ
    Else:
      send message (cheated, ours, theirs, Bob) to ℰ

Upon receiving (letsTrade, theirs, ours) from Bob:
  Send (willWeCheat, theirs, ours, Bob) to ℰ
  Wait for response from ℰ
  If response is (doNotCheat, theirs, ours, Bob):
    If transfer(ours, Bob) == True:
      Send (transferred, ours, Bob) to Bob and ℰ

transfer(object, receiver):
  If isAvailable(object):
    If object is money:
      send (pay, object, receiver) to F_Ledger
      wait for response from F_Ledger
      Upon receiving (paymentDone, object, receiver):
        return True
    Else: # object is asset
      send (transfer, object, receiver) to F_Assets
      wait for response from F_Assets
      Upon receiving (transferDone, object, receiver):
        return True
  return False

isAvailable(object):
  If object is money:
    send (doIhaveBalance, object) to F_Ledger
    wait for response from F_Ledger
    return response
  Else: # object is asset
    Send (doIHave, object) to F_Assets
    wait for response from F_Assets
    If response == (youHave, object):
      return True
    Else:
```

```
      return False

Upon receiving message (obtain, s) from 𝓔:
  send message (add, s) to 𝓕_Assets
  wait for response from 𝓕_Assets
  If response == (added, s)
    send message (obtained, s) to 𝓔
  Else
    send message (notObtained, s) to 𝓔

Upon receiving message (lose, s) from 𝓔:
  send message (remove, s) to 𝓕_Assets
  wait for response from 𝓕_Assets
  If response == (removed, s)
    send message (lost, s) to 𝓔
  Else
    send message (notLost, s) to 𝓔
```

$\mathcal{F}_{\text{SAT}}$

```
# Commented lines are the weak version, ⟨𝓕_SAT⟩.
Initialisation:
  ∀Alice ∈ 𝒫,
    util(Alice) = ⊥
  # advBlacklistedBy = ∅

Upon receiving (type, t) from Alice:
  util(Alice) = t

# TODO: define dist(normalresponse, adversarialresponse)
# TODO: be more flexible than grim trigger (gossip cheats,
    regain trust)
Upon receiving (satisfy, d, vendorList) from Alice:
  If util(Alice) == ⊥:
    send message (utilityNotSet) to Alice
    go to Idle state
  find offerList = {(Bob, x, s) ∈ vendorList × ℝ × Assets :
    s ∈ assets(Bob) and s ∈ d and x ≥ 0 and
    𝒢_Ledger.balance(Alice) ≥ x and
    util(Alice)(𝒢_Assets.possessions(Alice) ∪ {s},
        𝒢_Ledger.balance(Alice) − x) >
```

```
    util(Alice)(G_Assets.possessions(Alice), G_Ledger.balance(Alice))
    and
    util(Bob)(G_Assets.possessions(Bob) \ {s},
        G_Ledger.balance(Bob) + x) >
    util(Bob)(G_Assets.possessions(Bob), G_Ledger.balance(Bob))
  # send (createOfferList, d, vendorList, Alice) to A
  # wait for response from A
  # If response is a valid list of sellers, prices, and
      assets:
  #   punish A distList(offerList, response)
  #   offerList = response
  # If Alice ∈ advBlacklistedBy:
  #   remove from list all entries with A as seller
  # send (chooseBestSeller, d, offerList, Alice) to A
  # wait for response from A
  # If response ≠ (bestSeller, offerList, Bob, x, s),
      Bob ∈ P, x ∈ Money, s ∈ Asset:
    # e.g. Bob == ⊥
  (Bob, x, s) =
        argmax   {util(Alice)(G_Assets.possessions(Alice) ∪ {s},
      (Bob,x,s)∈list
        G_Ledger.balance(Alice) − x)}
  # Else:
  #   parse response as (bestSeller, offerList, Bob, x, s)
  #   (Bob', x', s') =
        argmax   {util(Alice)(G_Assets.possessions(Alice) ∪ {s},
      (Bob,x,s)∈list
  #       G_Ledger.balance(Alice) − x)}
  #   punish A distSeller((Bob, x, s), (Bob', x', s')) payoff
      reduction
  send (pay, Alice, Bob, x) to G_Ledger as Alice
  # If Bob == A:
  #   send (cheat, offerList, response, Alice) to A
  #   wait for response from A
  #   If response == yes:
  #     advBlacklistedBy = advBlacklistedBy ∪ Alice (grim
      trigger)
  #     send message (cheated, d, L) to Alice
  #     go to idle state
  send (give, Bob, Alice, s) to G_Assets as Bob
  send message (satisfied, d, L) to Alice
```

**Designer's (RPD [1]) utility:**

$$\forall p \in \mathcal{P} \text{ let event } E_{u,p} : \mathcal{Z} \text{ sends message } (\texttt{payoffs}, u_p : 2^A \times \mathbb{N}^* \to \mathbb{R}) \text{ at time 0 such that } \forall t \in \mathbb{N}^*, b, b' \in 2^A, b' \subseteq b, \text{ it is } u_p(b', t) \leq u_p(b, t) \text{ and } \forall \text{ sequences of bundles } B, \sum_{t \in \mathbb{N}^*} u(B_t, t) \in \mathbb{R} \ .$$

$$\forall p \in \mathcal{P}, b \in 2^A \text{ let event } E_{p,b,t} : \mathcal{F}_{\text{Assets}}.\texttt{assets}(Alice) = b \text{ at time } t$$

$$v^{\mathtt{D}}_{\langle \mathcal{F} \rangle, \mathcal{S}, \mathcal{Z}} = \sum_{p \in \mathcal{P}} \Pr_{\text{messages}}[E_{u,p}] \sum_{t \in \mathbb{N}^*} \sum_{b \in 2^A} \Pr_{\text{messages}}[E_{p,b,t}] \, u_p(b, t)$$

**Attacker's (RPD [1]) utility:**

$$v^{\mathtt{A}}_{\langle \mathcal{F} \rangle, \mathcal{S}, \mathcal{Z}} = \Pr_{\text{messages}}[E_{\mathcal{A}}] \sum_{t \in \mathbb{N}^*} \sum_{b \in 2^{\text{Assets}}} \Pr_{\text{messages}}[E_{\mathcal{A},b,t}] \, u_{\mathcal{A}}(b, t) - \langle F_{SAT} \rangle \text{ punishment (l. 43)}$$

## 5 Formal description of execution

The control function $C$ has the following information hardcoded:

- Duration of the game, $r_{\text{end}}$ (not disclosed to anyone else)
- Number of parties, $n$ (not known by honest parties)
- Set of (distinct) assets $A$

$C$ imposes the following rules on the execution:

- Up to one party is allowed to be corrupted at $r = 0$.
- $\mathcal{Z}$ is obliged to send one `(payoff,` $u : 2^A \times \mathbb{N}^* \to \mathbb{R}$`)` input and one `(endowment,` $b_P$`)` ($b_P \subseteq A$) to each player $P$ at $r = 0$. $C$ ensures that the bundles $b_{P_1}, \ldots, b_{P_n}$ are a partition of $A$. (TODO: We will later specify whether the utility functions and endowments are defined by $\mathcal{Z}$ or are given as input to $\mathcal{Z}$ at the beginning of the execution. We also have to specify whether the utility functions are oracles or PPT.)
- Each player is allowed to satisfy up to one desire on each $r > 0$. $C$ demands that $\mathcal{Z}$ provide exactly one `(satisfy,` $d$`)` ($d \subseteq A$) input to each player on each $r > 0$. The players must be activated in a round-robin fashion, with $\mathcal{A}$ being last. (Rushing) (TODO: how to hide which player is corrupted?)

$$P\text{'s messages for round } i : ms_i^P = \left(m_1, \ldots, m_{|ms_i^P|}\right)$$

$$\text{VIEW}_{\Pi,\mathcal{A},\mathcal{Z}}^{P,t_{\text{end}},n}(z) = \left( \underbrace{x_1^P, \ldots, x_{r_{\text{end}}}^P}_{\text{inputs by } \mathcal{Z}}, \underbrace{r^P}_{P\text{'s random tape}}, \underbrace{ms_1^P, \ldots, m_{r_{\text{end}}}^P}_{\text{messages received by } P} \right)$$

$$\text{VIEW}_{\Pi,\mathcal{A},\mathcal{Z}}^{t_{\text{end}},n}(z) = \langle \text{VIEW}_{\Pi,\mathcal{A},\mathcal{Z}}^{P,t_{\text{end}},n}(z) \rangle_{P \in \mathcal{P}}$$

## 6 $\mathcal{F}_{\text{SAT}}$ and $\Pi_{\text{SAT}}$ are potentially distinguishable

Consider the hybrid world of Fig. 1 (right) with $n$ ITMs executing $\Pi_{\text{SAT}}$, where $\mathcal{F}_{\text{Trust}}$ is replaced by $\mathcal{F}'_{\text{Trust}}$:

$\mathcal{F}'_{\text{Trust}}$

```
1 Upon receiving (chooseBestSeller, d, L) from Alice:
2   Bob ←R L ∪ {⊥}
3   send message (bestSeller, d, L, Bob) to Alice
```

We will show here that $\mathcal{E}$ can distinguish between $\mathcal{F}_{\text{SAT}}$ and $\Pi_{\text{SAT}}^{\mathcal{F}'_{\text{Trust}}}$.

*Distinguishability.* Consider the following adversary and environment:

$\mathcal{A}$

```
1 Upon receiving (chooseBestSeller, d, list, Alice):
2   If |list| ≠ 1 ∨ |d| ≠ 1:
3     go to Idle State
4   Bob = p : (p, x, s) ∈ list
5   s = asset : asset ∈ d
6   return (bestSeller, list, Bob, 1, s)
```

$\mathcal{E}$ `distinguisher`

```
1 Alice ←R P
2 Bob ←R P \ {Alice}
3 U_Alice(t, x, a) = x + 2|a|
4 U_Bob(t, x, a) = 2x + |a|
5 ∀p ∈ {Alice, Bob}:
6   send message (type, U_p) to p
7 s ←R Asset
```

```
8  send message (obtain, 1, Alice) to G_Ledger
9  send message (obtain, s, Bob) to G_Assets
10 send message (satisfy, {s}, {Bob}) to Alice
11
12 Upon receiving message (x, {s}, {Bob}) from Alice:
13   If x == satisfied:
14     return 1 # functionality
15   Else: # if x ∈ {cheated, unsatisfied}
16     return 0 # protocol
```

Because of the way $\mathcal{E}$ is built, there always exists a seller (*Bob*, line 2) who has an asset (line 9) that can satisfy the desire (line 10) of the buyer (*Alice*, line 1).

In case $\mathcal{E}$ interacts with $\mathcal{F}_{\text{SAT}}$, let $\mathcal{S}$ simulator that tries to simulate $\mathcal{A}$. $\mathcal{F}_{\text{SAT}}$ will always send the message `(chooseBestSeller,` $\{s\}$`,` $\{(Bob, 1, s)\}$`,` *Alice*`)` to $\mathcal{S}$ because:

1. *Alice* has one coin according to $\mathcal{E}$, line 8, as required by $\mathcal{F}_{\text{SAT}}$, line 18.
2. It is in *Alice*'s benefit for the trade to go through, since she values acquiring one asset more than one coin (`util(`*Alice*`)(`$\{s\}, 0$`)` $= 2 >$ $1 =$ `util(`*Alice*`)(`$\emptyset, 1$`)` as can be seen in $\mathcal{E}$, lines 3 and 6), as required in $\mathcal{F}_{\text{SAT}}$, line **??**.
3. It is in *Bob*'s benefit for the trade to go through, since he values acquiring one coin more than one asset (`util(`*Bob*`)(`$\{s\}, 0$`)` $= 1 <$ $2 =$ `util(`*Bob*`)(`$\emptyset, 1$`)` as can be seen in $\mathcal{E}$, lines 4 and 6), as required in $\mathcal{F}_{\text{SAT}}$, line **??**.

$\mathcal{S}$ should always match the buyer and the seller because of the way $\mathcal{A}$ is built. More precisely, $\mathcal{S}$ must always respond to `(chooseBestSeller,` $\{s\}$`,` $\{(Bob, 1, s)\}$`, _)` with `(bestSeller,` $\{(Bob, 1, s)\}$`,` *Bob*`, 1,` $s$`)` in order to correctly simulate $\mathcal{A}$ (lines 4-6).

Furthermore, $\mathcal{F}_{\text{SAT}}$ never cheats on a trade and always chooses a suitable seller, price and asset (given that there exists one, which is the case here as we saw earlier) (lines 35-53), thus the exchange will always complete correctly and $\mathcal{E}$ will receive `satisfied` as response. $\mathcal{E}$ will always correctly output 1 (which corresponds to the functionality, line 14).

On the other hand, in case $\mathcal{E}$ interacts with $\Pi_{\text{SAT}}$, then we observe that $\mathcal{F}'_{\text{Trust}}$ does not choose players depending on their reputation (line 2), thus in this particular setting the utility of the players does not depend on their reputation. Thus, if $\mathcal{F}'_{\text{Trust}}$ does not respond with $\perp$, it is always in *Bob*'s interest to cheat, since keeping the asset is preferrable to giving it ($\mathcal{E}$, lines 4 and 6). Thus *Alice*'s response to $\mathcal{E}$ will always be `cheated`.

If $\mathcal{F}'_{\text{Trust}}$ responds with $\perp$ (line 2), the trade will not go through ($\Pi_{\text{SAT}}$, lines 33-34) and $\mathcal{E}$ will receive `unsatisfied` as a response. In all cases $\mathcal{E}$ will correctly output 0 (which corresponds to the protocol, line 16). □

$\mathcal{F}_{\text{Trust}}$

1 Has oracle access to every player's *Alice* utility function $U_{Alice}$

2

3 Upon receiving (chooseBestSeller, $d$, $L$) from *Alice* at time $t$:

4 $\mathtt{m}_{\text{Alice}}$ = $\mathcal{G}_{\text{Ledger}}$.money(*Alice*)

5 $\mathtt{a}_{\text{Alice}}$ = $\mathcal{G}_{\text{Assets}}$.assets(*Alice*)

6 $\mathtt{m}_{\text{Bob}}$ = $\mathcal{G}_{\text{Ledger}}$.money(*Bob*)

7 $\mathtt{a}_{\text{Bob}}$ = $\mathcal{G}_{\text{Assets}}$.assets(*Bob*)

8 *Bob* = $\underset{Bob \in L}{\text{argmax}} \{ U_{Alice}(\mathtt{t}, \mathtt{m}_{\text{Alice}} - x, \mathtt{a}_{\text{Alice}} \cup \{\mathtt{s}\}) :$

9 $\mathtt{s} \in \mathtt{d} \cap \mathtt{a}_{\text{Bob}} \wedge \mathtt{m}_{\text{Alice}} \geq \mathtt{x} \wedge$

10 $U_{Alice}(\mathtt{t}, \mathtt{m}_{\text{Alice}} - \mathtt{x}, \mathtt{a}_{\text{Alice}} \cup \{\mathtt{s}\}) > U_{Alice}(\mathtt{t}, \mathtt{m}_{\text{Alice}}, \mathtt{a}_{\text{Alice}}) \wedge$

11 $U_{Bob}(\mathtt{t}, \mathtt{m}_{\text{Bob}} + x, \mathtt{a}_{\text{Bob}} \setminus \{\mathtt{s}\}) > U_{Bob}(\mathtt{t}, \mathtt{m}_{\text{Bob}}, \mathtt{a}_{\text{Bob}}) \wedge$

12 $U_{Bob}(\mathtt{t}, \mathtt{m}_{\text{Bob}} + x, \mathtt{a}_{\text{Bob}}) < U_{Bob}(\mathtt{t}, \mathtt{m}_{\text{Bob}}, \mathtt{a}_{\text{Bob}}) \}$ # impossible

13 send message (bestSeller, d, L, *Bob*) to *Alice*

Assumptions:

- Trades are atomic. $\mathcal{F}_{\text{Trust}}$ can deterministically decide whether *Bob* will complete the trade.
- The internal workings of $\mathcal{F}_{\text{Trust}}$ is common knowledge to the players. Their utility depends on it.

Note: It would be interesting to see how utilities (as algorithms) and $\mathcal{F}_{\text{Trust}}$ have a "fixed point".

$\Pi_{\text{Trust}}$

1 Upon receiving (chooseBestSeller, $d$, $L$) from *Alice*:

2 *Bob* =

3 $\underset{(Bob,x) \in L}{\text{argmax}} \{\mathcal{O}_{\text{Rep}}(Alice, Bob, s)$util(*Alice*)($state_{Alice} \cup s - x$) +

4 $(1 - \mathcal{O}_{\text{Rep}}(Alice, Bob, s))$util(*Alice*)($state_{Alice} - x$) :

5 $s \in d \wedge s \in$ assets(*Bob*) $\wedge$ *Alice* has $x$ coins $\}$

6 # drop "Bob has s"?

7 If util(*Alice*)($state_{Alice}$) $\geq$ util(*Alice*)($state_{Alice} \cup s - x$):

8 *Bob* = $\perp$

9 send message (bestSeller, $d$, $L$, *Bob*) to *Alice*

## 7 An open source reputation management algorithm is impossible

**Theorem 1.** *Let* $U_{Alice} : Money \times multiset\,(Assets) \times Rep \to \mathbb{R}$*, where* $Rep \in \mathbb{N}$*. (*$\mathcal{F}_{\text{Trust}}$ *does not have access to any oracle.) Then* $\nexists \mathcal{F}_{\text{Trust}} : \Pi_{\text{SAT}}$ *UC-realizes* $\mathcal{F}_{\text{SAT}}$*.*

*Proof.* We will prove this by contradiction. Let $\mathcal{F}_{\text{Trust}}$ be such a functionality. If it does not allow for cheats (grim trigger for seller on buyer reporting a cheat), then a malicious buyer (also prospective seller) can try buying from a competitor, falsely report a cheat and beat the competition (thus increasing their utility). If it allows for cheats, then it does not emulate $\mathcal{F}_{\text{SAT}}$. TODO: complete □

**Definition 1 (Cheating Security).** *We say that a protocol is secure against cheating if* $\forall$ *PPT* $\mathcal{E}, \Pr[\mathcal{E}$ *receiving* $(\texttt{cheated}, \_, \_)]$ *is negligible.*

Question: how to express security against false $(\texttt{cheated}, \_, \_)$ messages?

## 8 A case against honest and malicious parties

Satisfying Definition 1 in case there exist honest parties and an Adversary seems impossible. When an honest buyer pays a malicious seller, then the malicious seller can always cheat and the honest buyer will report it to $\mathcal{E}$. It seems improbable that we can create an $\mathcal{F}_{\text{Trust}}$ that returns a malicious party with negligible probability, especially since $\mathcal{F}_{\text{Trust}}$ cannot know from the outset which parties are corrupted and which are not.

Similarly, if a malicious buyer trades with an honest seller, then the seller will go through with the trade but the buyer will be able to falsely report a cheat to $\mathcal{E}$. This as well cannot be avoided, since $\mathcal{E}$ does not know whether the player to which it sends a $(\texttt{satisfy}, \texttt{d}, \texttt{L})$ message is corrupted.

Nevertheless, if we completely change perspective and have all players be rational, then it is possible that for certain utility functions no player will cheat or report false cheating for fear of future punishment.

## 9 Interaction with $F_{\text{Trust}}$

*Alice* receives an offer from *Bob* for price $p$. If $Trust_{Alice \to Bob} \geq p$, then she pays Bob by telling $F_{\text{Trust}}$ to redistribute the $p$ of her direct trust to him so that we end up with $Trust_{Alice \to Bob} = DirectTrust_{Alice \to Bob}$.

If the trade completes correctly and *Bob* takes $p'$ coins from *Alice*'s direct trust, where $0 \leq p' \leq p$, *Alice* undoes the initial redistribution and on top of this she adds another $p - p'$ as direct trust to *Bob*. If she has spare exclusive coins, she uses them, otherwise she decreases her direct trust to the least recently used direct trust. LRU direct trust is the oldest direct trust that helped her make a decision — for that she has to locally timestamp the direct trusts that are leveraged whenever she decides to trust someone.

If the trade fails to complete and *Bob* takes $p'$ coins from *Alice*'s direct trust, where $0 \leq p' \leq p$, she takes her $p - p'$ coins back and also reduces her trust to the players towards whom she had direct trusts that pointed her to *Bob* by another total $p$, keeping this money for exclusive use.

How can *Bob* exploit this strategy? He would need the collaboration of the players *Alice* trusts directly. They have to steal from her at the same time as he does. If we assume adaptive corruptions, then this is rather easy.

*Bob* seems to be able to cheat on half of the trades and still he will have some incoming direct trust. If he has say $100p$ incoming direct trust, then he can maintain the same balance by selling honestly one item of value $p$ and then cheating on the next trade of the same item alternatively. It is not directly obvious that this strategy will eventually leave *Bob* alone — indeed, he may build a lucrative mafia-like scheme this way; he just has to find constantly new first-line cheaters.

In order to mitigate this attack, a "know your customer" scheme may be necessary. Indeed, if honest sellers only sell to *Bob* only if they trust him enough, this scheme does not work because nobody will accept his money. Exclusive coins will only be useful as reserve to be directly trusted to others. This however completely overhauls the way money works today...

## 10 Single Corruption Game

## 11 Market Description as Bayesian game

$G = \left(N, (A_i)_{i \in N \cup \{0\}}, (T_i)_{i \in N}, (u_i)_{i \in N}, p\right)$
$N = \{1, \ldots, n\}$ for some $n \in \mathbb{N}$
$A_0 = (i, d)^* \xleftarrow{r} (N \times D)^*$
$\forall i \in N, A_i \ldots$
$\forall i \in N, T_i$ probability distribution over $\mathbb{N}$:

**Algorithm 1** singleCorruptionGame$_{\mathcal{A}}(G=(V,E,W),CP,D,w,end)$

**for all** $v \in V$ **do**
  **if** $v \neq w$ **then**
    start $\Pi_{\mathrm{SAT}}$ on behalf of $v$
  **else**
    start $\mathcal{A}(v)$
  **end if**
  $U(v) \leftarrow CP$
  send (`utility`,$U(v)$) to $v$
**end for**
**for** t $\leftarrow 1$ to *end* **do**
  $v \xleftarrow{r} V$; $d \xleftarrow{r} D$; $L \overset{r}{\subseteq} V$
  send (`satisfy`,$d$,$L$) to $v$
  wait for (`satisfied`,$d$,$L$) or (`cheated`,$d$,$L$) from $v$
**end for**
output $U(w)$

$$\forall i \in N, T_i : \mathbb{N} \to [0,1] \text{ with } \sum_{t=0}^{\infty} T_i(t) = 1$$

$$\forall i \in N, u_i : \sum_{t=0}^{\infty} T_i(t) \frac{sat(i,t,D)}{|D|}$$

$p$ probability distribution over types:

$$p : T_1 \times \cdots \times T_n \to [0,1] \text{ with } \sum_{t \in T_1 \times \cdots \times T_n} p(t) = 1$$

## References


1. Garay J., Katz J., Maurer U., Tackmann B., Zikas V.: Rational protocol design: Cryptography against incentive-driven adversaries. In Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on: pp. 648–657: IEEE (2013)