Online Pseudonym Parties, a proof-of-unique-human system

Johan Nygren, johanngrn@gmail.com

ABSTRACT: Online Pseudonym Parties is based on pseudonym events, global and simultaneous verification events that occur at the exact same time for every person on Earth. In these events, people are randomly paired together, 1-on-1, to verify that the other is a person, in a pseudo-anonymous context, over video chat. The event lasts 15 minutes, and the proof-of-unique-human is that you are with the same person for the whole event. The proofs, valid for a month, can be disposed of once no longer valid, and are untraceable from month to month.

Introduction, or What is it like to be a nym?

Online Pseudonym Parties is a population registry for a new global society that is built on top of the internet. It provides a simple way to give every human on Earth a proof-of-unique-human, and does so in a way that cannot exclude or reject a human being, as long as the average person in the population would recognize them as human. This means it is incapable of shutting anyone out, and that it will inherently form a single global population record, that integrates every single human on Earth. It is incapable of discrimination, because it is incapable of distinguishing one person from another, since it has absolutely zero data about anyone. It's incapable of discriminating people by gender, sexuality, race or belief. The population registry also has infinite scalability. The pairs, dyads, are autonomous units and the control structure of Online Pseudonym Parties. Each pair is concerned only with itself, compartmentalized, operating identically regardless of how many times the population doubles in size.

The "court" system, and the "virtual border"

The key to Online Pseudonym Parties is the "court" system, that subordinates people under a random pair, to be verified in a 2-on-1 way. This is used in two scenarios, both equally important. The first, is when a person is paired with an account that does not follow protocol. In most cases, this would be a computer script, or a person attempting to participate in two or more pairs at the same time. Normally, in the pairs, people have to mutually verify each other to be verified. In case a person is paired with an attacker, they can choose to break up their pair, and subordinate both people in the pair under a random pair each, a "court". The attacker, if they were a bot, will not be verified by the court, or have any ability to coerce it, while the normal person will be verified by their "court". Both people in the pair that "judges" a court have to verify the person being judged.

```
struct Court { uint id; bool[2] verified; }

mapping (uint => mapping (address => Court)) public court;
mapping (uint => mapping (uint => address)) public courtIndex;
mapping (uint => uint) public immigrants;

function dispute() external {
    uint t = schedule()-2;
    uint id = nym[t][msg.sender].id;
    require(id != 0);
    require(!pairVerified(t, getPair(id)));
    pair[t][getPair(id)].disputed = true;
}
```

The second scenario for the "court" system, is the "virtual border". The population has a form of "border" or "wall" around it, and anyone not verified in the previous event is on the "outside" of this "border". To register, you need to go through an "immigration process". During this process, you are assigned to a "court", another pair, and they verify you in a 2-on-1 way, so that a bot would have no way to intimidate or pressure this "border police". This "border", together with the dispute mechanism, acts as a filter that prevents any attackers to the system.

2, 4, 8, 16… 1 billion, growth by doubling

The pairs are the control structure of Online Pseudonym Parties, and each pair is concerned only with itself, regardless of how many times the population doubles in size. The population is allowed to double in size each event, made possible by that each person verified during an event is authorized to invite another person to the next event. This allows the population to grow from 2 to 10^3 (1024) in 10 events, 10^3 to 10^6 (1048576) in 20 events, and 10^6 to 10^9 (1073741824) in 30 events. There is no theoretical upper limit to this scalability mechanism, Online Pseudonym Parties has infinite scalability.

```
function initialize(bytes32 _commit) external {
    uint t = schedule();
    require(pairs(t-2) == 0 && pairs(t) == 0);
    shuffler[t].push(msg.sender);
    commit[t][msg.sender] = _commit;
    nym[t][msg.sender].status = Status.Commit;
}
```

Randomization, Vires in Numeris

The randomization of the pairs is a major security assumption, and is very simple. The processes are separated over two phases. In the first phase, people register and are appended to a list. In the second phase, the list is shuffled using a randomly generated seed. The population is able to autonomously generate the random seeds that secure the system, using “random majority vote”, described in the next section.

```
uint entropy;

enum Status { Inactive, Commit, Party }
struct Nym { uint id; Status status; }

mapping (uint => mapping (address => Nym)) public nym;
mapping (uint => address[]) public shuffler;
mapping (uint => uint) public shuffled;

function register(bytes32 _commit) external {
    uint t = schedule();
    require(!halftime(t));
    deductToken(Token.Registration, t);
    shuffler[t].push(msg.sender);
    commit[t][msg.sender] = _commit;
    nym[t][msg.sender].status = Status.Commit;
}
function shuffle() external {
    uint t = schedule()-1;
    uint _shuffled = shuffled[t];
    if(_shuffled == 0) entropy = generator[t][leader[t]];
    uint unshuffled = registered(t) -_shuffled;
    if(unshuffled > 0) {
        uint randomNumber = entropy % unshuffled;
        entropy ^= uint160(shuffler[t][randomNumber]);
        (shuffler[t][unshuffled-1], shuffler[t][randomNumber]) = (shuffler[t][randomNumber],
shuffler[t][unshuffled-1]);
        nym[t][shuffler[t][unshuffled-1]].id = unshuffled;
        shuffled[t]++;
    }
}
```

The population generates random numbers

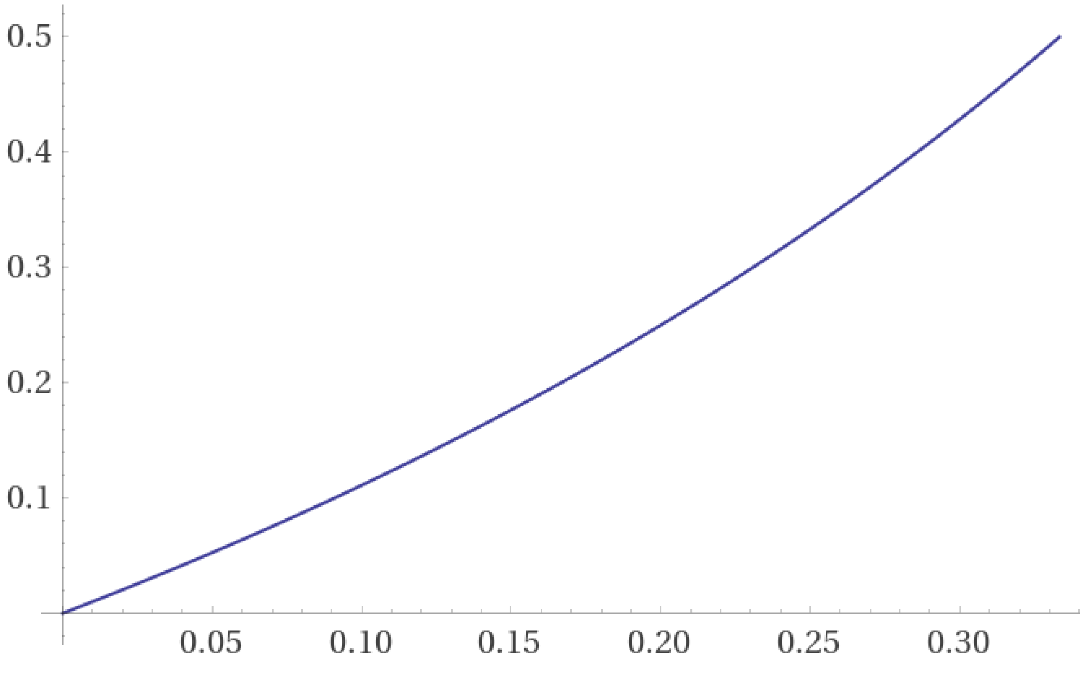
The population has a protocol that allows them to generate a random numbers each period in a way that cannot be manipulated by foreign actors. It relies on a form of majority vote, but, no one can control what they vote for. They can only control that their vote is random, and know that the vote of every other person is random. This is possible by using a commit-reveal scheme, where votes are committed before the votes in the previous period have been revealed. The random number generated by majority vote each period, is used to “mutate” the votes in the next period. Votes contribute randomness to generators. There are as many generators as there are pairs, and the probability that some generator gets k “hits” is 2^k/(e^2 k!). The generator that gets the most “hits”, i.e., a majority vote, wins.

```
mapping (uint => mapping (address => bytes32)) public commit;

mapping (uint => mapping (uint => uint)) public generator;
mapping (uint => mapping (uint => uint)) public points;
mapping (uint => uint) public leader;
```

Collusion attacks

Online Pseudonym Parties is vulnerable to one type of attack vector only, collusion attacks. The success of collusion attacks increases quadratically, as x^2, where x is the percentage colluding. Repeated attacks conform to the recursive sequence $a[n] == (x + a[n—1])^2/(1 + a[n—1])$, and can be seen to approach n as x -> 1. It plateaus at the limit $a[\infty] = x^2/(1-2x)$ for $0<x<0.5$. Colluders reach 50% control when $(a[\infty] + x) == (1+a[\infty])/2$, this happens at x = 1/3, i.e., Online Pseudonym Parties is a 66% majority controlled system.



plot (x^2/(1-2x)+x)/(1+x^2/(1-2x)) from 0 to 1/3 | Computed by Wolfram|Alpha

“Border attack” component of collusion attacks

The “border attack” is when x% of the population collude, and get pairs with two colluders for x% of them, can get x% of the immigrants they can invite into those pairs, giving them $p*x\%^3$ bots, on top of $p*x\%^2$ for the collusion attack, in total $(x\%^3 + x\%^2)*p$, 1+x% times than with just collusion attack. This attack shifts the curve for when colluders gain 50% control, 50% of all proof-of-unique-human, to 22% instead of 33%. 2/3rds majority requirement is already a bit high, so this would not be good. This attack is 100% defendable, by adding one rule. To remove the “border attack” completely, require that people sacrifice their ability to register for the next event to claim ability to invite someone new, and give them ability to invite two people (one being themselves.) If colluders attempt to attack, the population can respond by not verifying any immigrants, taking a massive hit at the colluders who had to invest $1/x\%^2$ more accounts than they can get bots, i.e., get $x\%^2$ accounts for every account they lose. This breaks that attack vector completely. It does not work during “growth” period of system, so it has to be a mode that is activated once population has reached full size.

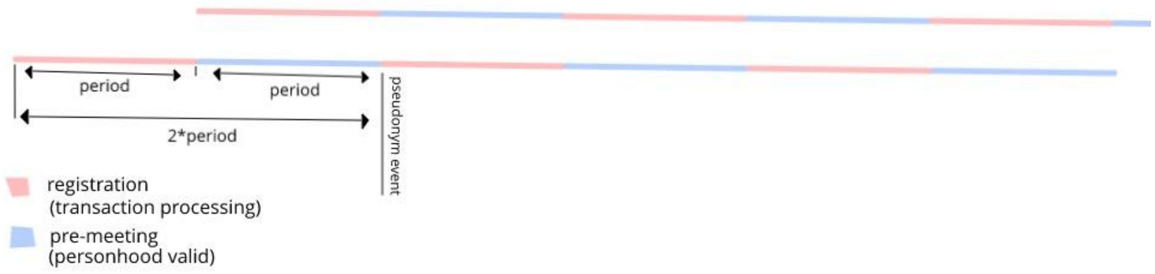
```
bool borderSetting;

function invite() external {
    require(borderSetting == true);
    uint t = schedule();
    require(balanceOf[t][Token.Registration][msg.sender] >= 1);
    balanceOf[t][Token.Registration][msg.sender]--;
    balanceOf[t][Token.Immigration][msg.sender]+=2;
}
```

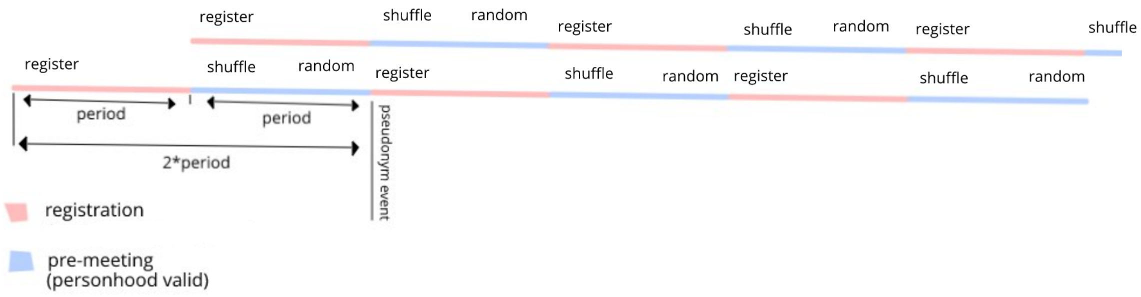
Man-in-the-middle attacks, and “pre-meetings”

Man in the middle attacks are when two fake accounts relay the communication between the two real humans the fake accounts are chosen to verify. Then the two real humans each verified a bot. These are defended against simply by the real people asking each other what pair they are in. To have extra security, they can be defended in a way that is as “Turing safe” (same difficulty for breaking Turing test) as the actual event itself, but it requires an extra step. The ideal defense adds a "handshake" to secure the video channel. The mechanism for this, the pair schedules a “pre-meeting” at a random time before the event, by agreeing on a random number using a commit-reveal scheme. The time for the pre-meeting is the sum of both numbers. Public keys are exchanged along with the numbers, as part of the encrypted message. Once they have proven encrypted numbers have been exchanged (either by meeting over video while doing the exchange, or, having a deadline), they then meet at the agreed time. This “pre-meeting” can only take place if they got the same number. This proves that their channel is secure by validating that they both got the same number, and that the public key of the other person (that was also exchanged) is the public key they are paired with. The probability of the man in the middle attacking this in a way that both peers get the same number is 1/numberSize. This defense is as "Turing safe" as the actual event itself.

To allow time for “pre-meetings” while also having enough time to process transactions for registration, “random majority vote”, and shuffling, two separate pseudonym event “timelines” can exist, allowing an entire period for “pre-meetings”. See image below.



With the random majority vote and shuffle:



Scheduling the pseudonym event

Scheduling is trivial. The current month is calculated using a timestamp for the genesis event, the periodicity in seconds, and the current time.

```
uint constant period = 4 weeks;
uint constant genesis = 198000;

function schedule() public view returns(uint) { return ((block.timestamp - genesis) / period); }
function toSeconds(uint _t) public pure returns (uint) { return genesis + _t * period; }
```

An anonymous population registry

Since “who a person is” is not a factor in the proof, mixing of the proof-of-unique-human does not reduce the reliability of the protocol in any way. It is therefore allowed, and encouraged. This is practically achieved with “tokens” that are intermediary between verification in one pseudonym event and registration for the next event, authorizing mixer contracts to handle your “token” using the approve() function.

```
enum Token { Personhood, Registration, Immigration }

mapping (uint => mapping (Token => mapping (address => uint))) public balanceOf;
mapping (uint => mapping (Token => mapping (address => mapping (address => uint)))) public allowed;

function _transfer(uint _t, address _from, address _to, uint _value, Token _token) internal {
    require(balanceOf[_t][_token][_from] >= _value);
    balanceOf[_t][_token][_from] -= _value;
    balanceOf[_t][_token][_to] += _value;
}
function transfer(address _to, uint _value, Token _token) external {
    _transfer(schedule(), msg.sender, _to, _value, _token);
}
function approve(address _spender, uint _value, Token _token) external {
    allowed[schedule()][_token][msg.sender][_spender] = _value;
}
function transferFrom(address _from, address _to, uint _value, Token _token) external {
    uint t = schedule();
    require(allowed[t][_token][_from][msg.sender] >= _value);
    _transfer(t, _from, _to, _value, _token);
    allowed[t][_token][_from][msg.sender] -= _value;
}
```

References

Pseudonym Parties: An Offline Foundation for Online Accountable Pseudonyms, https://pdos.csail.mit.edu/papers/accountable-pseudonyms-socialnets08.pdf (2008)

Pseudonym Pairs: A foundation for proof-of-personhood in the web 3.0 jurisdiction, https://panarchy.app/PseudonymPairs.pdf (2018)

```
contract OnlinePseudonymParties {

    bool public borderSetting;

    uint constant public period = 4 weeks;
    uint constant genesis = 198000;
    mapping (uint => uint) public hour;

    function schedule() public view returns(uint) { return ((block.timestamp - genesis) / period); }
    function toSeconds(uint _t) public pure returns (uint) { return genesis + _t * period; }
    function halftime(uint _t) public view returns (bool) { return((block.timestamp > toSeconds(_t)+period/2)); }

    function computeHour(uint _t) public { hour[_t] =  1 + uint(keccak256(abi.encode(_t)))%24; }
    function pseudonymEvent(uint _t) public returns (uint) { if(hour[_t] == 0) computeHour(_t); return toSeconds(_t) + hour[_t]*1 hours; }

    uint entropy;

    mapping (uint => mapping (address => bytes32)) public commit;
    mapping (uint => mapping (uint => uint)) public generator;
    mapping (uint => mapping (uint => uint)) public points;
    mapping (uint => uint) public leader;

    enum Status { Inactive, Commit, Party }

    struct Nym { uint id; Status status; }
    struct Pair { bool[2] verified; bool disputed; }
    struct Court { uint id; bool[2] verified; }

    mapping (uint => mapping (address => Nym)) public nym;
    mapping (uint => address[]) public shuffler;
    mapping (uint => uint) public shuffled;
    mapping (uint => mapping (uint => Pair)) public pair;
    mapping (uint => mapping (address => Court)) public court;
    mapping (uint => mapping (uint => address)) public courtIndex;
    mapping (uint => uint) public immigrants;

    mapping (uint => uint) public population;
    mapping (uint => mapping (address => bool)) public proofOfUniqueHuman;

    enum Token { Personhood, Registration, Immigration }

    mapping (uint => mapping (Token => mapping (address => uint))) public balanceOf;
    mapping (uint => mapping (Token => mapping (address => mapping (address => uint)))) public allowed;

    function registered(uint _t) public view returns (uint) { return shuffler[_t].length; }
    function pairs(uint _t) public view returns (uint) { return (registered(_t)/2); }
    function deductToken(Token _token, uint _t) internal { require(balanceOf[_t][_token][msg.sender] >= 1); balanceOf[_t][_token][msg.sender]--; }
    function pairVerified(uint _t, uint _pair) public view returns (bool) { return (pair[_t][_pair].verified[0] == true && pair[_t][_pair].verified[1] == true); }
    function getPair(uint _id) public pure returns (uint) { return (_id+1)/2; }
    function getCourt(uint _t, uint _court) public view returns (uint) { require(_court != 0); return 1+(_court-1)%pairs(_t); }

    function register(bytes32 _commit) external {
        uint t = schedule();
        require(!halftime(t));
        deductToken(Token.Registration, t);
        shuffler[t].push(msg.sender);
        commit[t][msg.sender] = _commit;
        nym[t][msg.sender].status = Status.Commit;
    }
    function immigrate() external {
        uint t = schedule();
        require(!halftime(t));
        deductToken(Token.Immigration, t);
        immigrants[t]++;
        court[t][msg.sender].id = immigrants[t];
        courtIndex[t][immigrants[t]] = msg.sender;
    }
```

```
function shuffle() external {
    uint t = schedule()-1;
    uint _shuffled = shuffled[t];
    if(_shuffled == 0) entropy = generator[t][leader[t]];
    uint unshuffled = registered(t) -_shuffled;
    if(unshuffled > 0) {
        uint randomNumber = entropy % unshuffled;
        entropy ^= uint160(shuffler[t][randomNumber]);
        (shuffler[t][unshuffled-1], shuffler[t][randomNumber]) = (shuffler[t][randomNumber], shuffler[t][unshuffled-1]);
        nym[t][shuffler[t][unshuffled-1]].id = unshuffled;
        shuffled[t]++;
    }
}
function reveal(uint _entropy) external {
    uint t = schedule()-1;
    require(halftime(t+1));
    uint id = nym[t][msg.sender].id;
    require(id != 0);
    require(nym[t][msg.sender].status == Status.Commit);
    require(keccak256(abi.encode(_entropy)) == commit[t][msg.sender]);
    uint vote = ((_entropy%pairs(t)) + id)%pairs(t);
    generator[t][vote] ^= _entropy;
    points[t][vote]++;
    if(points[t][vote] > points[t][leader[t]]) leader[t] = vote;
    nym[t][msg.sender].status = Status.Party;
}

function verify() external {
    uint t = schedule()-2;
    require(block.timestamp > pseudonymEvent(t+2));
    require(nym[t][msg.sender].status == Status.Party);
    uint id = nym[t][msg.sender].id;
    require(id != 0);
    require(pair[t][getPair(id)].disputed == false);
    pair[t][getPair(id)].verified[id%2] = true;
}
function judge(address _account) external {
    uint t = schedule()-2;
    require(block.timestamp > pseudonymEvent(t+2));
    uint signer = nym[t][msg.sender].id;
    require(getCourt(t, court[t][_account].id) == getPair(signer));
    court[t][_account].verified[signer%2] = true;
}

function allocateTokens(uint _t, uint _pair) internal {
    require(pairVerified(_t-2, _pair));
    balanceOf[_t][Token.Personhood][msg.sender]++;
    balanceOf[_t][Token.Registration][msg.sender]++;
    if(borderSetting == false) balanceOf[_t][Token.Immigration][msg.sender]++;
}
function completeVerification() external {
    uint t = schedule()-2;
    require(nym[t][msg.sender].status != Status.Inactive);
    uint id = nym[t][msg.sender].id;
    allocateTokens(t+2, getPair(id));
    nym[t][msg.sender].status = Status.Inactive;
}
function courtVerdict() external {
    uint t = schedule()-2;
    require(court[t][msg.sender].verified[0] == true && court[t][msg.sender].verified[1] == true);
    allocateTokens(t+2, getCourt(t, court[t][msg.sender].id));
    delete court[t][msg.sender];
}
function claimPersonhood() external {
    uint t = schedule();
    deductToken(Token.Personhood, t);
    proofOfUniqueHuman[t][msg.sender] = true;
    population[t]++;
}
function invite() external {
    require(borderSetting == true);
    uint t = schedule();
    deductToken(Token.Registration, t);
    balanceOf[t][Token.Immigration][msg.sender]+=2;
}
```

```
    function dispute() external {
        uint t = schedule()-2;
        uint id = nym[t][msg.sender].id;
        require(id != 0);
        require(!pairVerified(t, getPair(id)));
        pair[t][getPair(id)].disputed = true;
    }
    function assignCourt(uint _court, uint _t) internal {
        uint _pairs = pairs(_t);
        _court = 1+(_court-1)%_pairs;
        uint i = 0;
        while(courtIndex[_t][_court + _pairs*i] != address(0)) i++;
        court[_t][msg.sender].id = _court + _pairs*i;
        courtIndex[_t][_court + _pairs*i] = msg.sender;
    }
    function reassignNym() external {
        uint t = schedule()-2;
        uint id = nym[t][msg.sender].id;
        require(pair[t][getPair(id)].disputed == true);
        assignCourt(uint256(uint160(msg.sender)) + id, t);
        delete nym[t][msg.sender];
    }
    function reassignCourt() external {
        uint t = schedule()-2;
        uint id = court[t][msg.sender].id;
        uint _pair = getCourt(id, t);
        require(pair[t][_pair].disputed == true);
        delete court[t][msg.sender];
        assignCourt(1 + uint256(uint160(msg.sender)^uint160(shuffler[t][_pair*2])^uint160(shuffler[t][_pair*2-1])), t);
    }

    function _transfer(uint _t, address _from, address _to, uint _value, Token _token) internal {
        require(balanceOf[_t][_token][_from] >= _value);
        balanceOf[_t][_token][_from] -= _value;
        balanceOf[_t][_token][_to] += _value;
    }
    function transfer(address _to, uint _value, Token _token) external {
        _transfer(schedule(), msg.sender, _to, _value, _token);
    }
    function approve(address _spender, uint _value, Token _token) external {
        allowed[schedule()][_token][msg.sender][_spender] = _value;
    }
    function transferFrom(address _from, address _to, uint _value, Token _token) external {
        uint t = schedule();
        require(allowed[t][_token][_from][msg.sender] >= _value);
        _transfer(t, _from, _to, _value, _token);
        allowed[t][_token][_from][msg.sender] -= _value;
    }

    mapping (uint => mapping(address => bool)) public votedBorderSetting;
    mapping (uint => uint) public borderVoteCounter;

    function voteOnBorder() external {
        uint t = schedule()-1;
        require(proofOfUniqueHuman[t][msg.sender] == true);
        require(votedBorderSetting[t][msg.sender] == false);
        borderVoteCounter[t]++;
        votedBorderSetting[t][msg.sender] = true;
    }
    function changeBorderSetting() external {
        uint t = schedule()-1;
        require(borderVoteCounter[t] > population[t]/2);
        borderSetting = !borderSetting;
        delete borderVoteCounter[t];
    }

    function initialize(bytes32 _commit) external {
        uint t = schedule();
        require(pairs(t-2) == 0 && pairs(t) == 0);
        shuffler[t].push(msg.sender);
        commit[t][msg.sender] = _commit;
        nym[t][msg.sender].status = Status.Commit;
    }
}
```