

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 The proofs, valid for a month, can be disposed of once no longer valid, and are untraceable from month to month.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;

function dispute() external {
    uint t = schedule()-1;
    uint _pair = getPair(nym[t][msg.sender].id);
    require(_pair != 0);
    require(!pairVerified(t, _pair));
    pair[t][_pair].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. The constructor allows the contract to initialize from two people, and can also be used when transferring the population from one ledger to another as digital ledgers get more advanced.

```
constructor(uint _population, address _genesis) {
    balanceOf[0][Token.Registration][_genesis] = _population;
}
```

Randomization, Vires in Numeris

The population is randomized by that each person who invokes register() will take the place of a random other person, and move that other person to the end. This mechanism keeps the computational cost per person low.

```
uint entropy;

function getRandomNumber() internal returns (uint) {
    return entropy = uint(keccak256(abi.encodePacked(blockhash(block.number - 1), entropy)));
}

struct Nym { uint id; bool verified; }

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

function register() public {
    uint t = schedule();
    deductToken(t, Token.Registration);
    uint id = 1;
    if(registered(t) != 0) {
        id += getRandomNumber() % registered(t);
        registry[t].push(registry[t][id-1]);
        nym[t][registry[t][id-1]].id = registered(t);
    }
    else registry[t].push();
    registry[t][id-1] = msg.sender;
    nym[t][msg.sender].id = id;
}
```

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 public genesis = 1631944800;
uint constant public period = 4 weeks;

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

The event is scheduled to always happen on the weekend, for all time zones. The exact hour of the event varies, to be fair to all time zones.

```
function hour(uint _t) public pure returns (uint) { return 1 + uint(keccak256(abi.encode(_t)))%24; }
function pseudonymEvent(uint _t) public pure returns (uint) { return toSeconds(_t) + hour(_t)*1 hours; }
```

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;
}
```

Proof-of-unique-human as a commodity

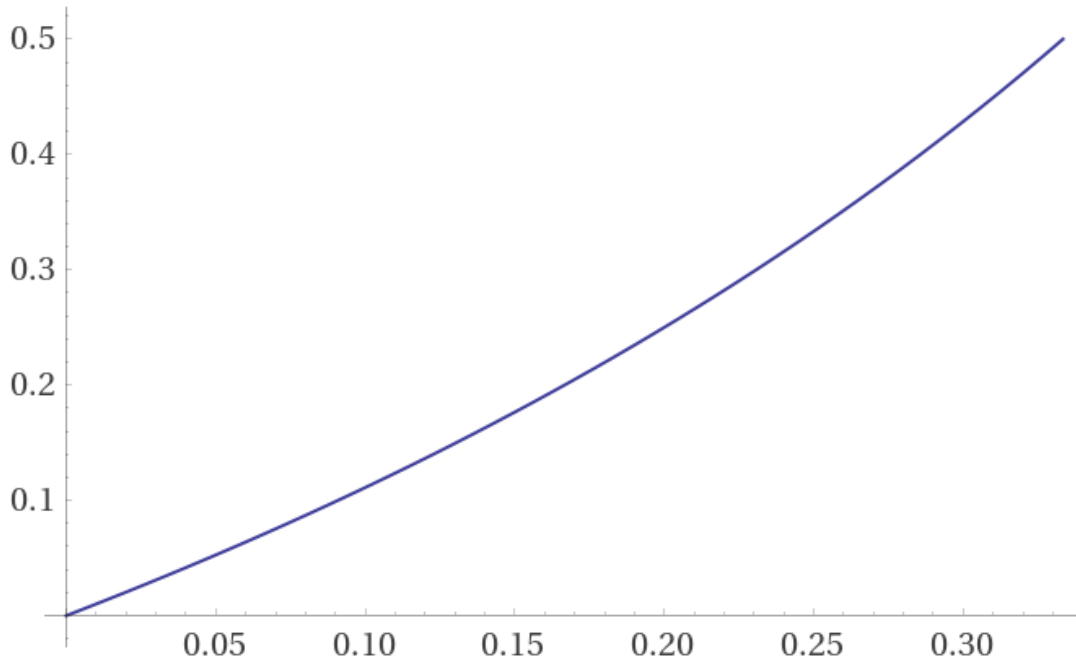
The proof-of-unique-human is only valid for one month, untraceable from month to month, and disposable once expired. The population registry has no concept of “who you are”. It cannot distinguish one person from another. Any other contract can build applications based on this proof-of-unique-human just by referencing proofOfUniqueHuman[_period][_account]. Applications that use some kind of majority vote, can reference population[_period] to know how many votes are needed to be >50% of the population.

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

function claimPersonhood() external {
    uint t = schedule();
    deductToken(t, Token.Personhood);
    proofOfUniqueHuman[t][msg.sender] = true;
    population[t]++;
}
```

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 $x \rightarrow 1$. 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

Very short summary, Overview

Everyone on Earth meets 1-on-1, as strangers, at the exact same time, for 15 minutes. Proof is valid a month, then new global event. Has a "virtual border" around it. If verified, can register again. If not, have to "immigrate", assigned under random pair, 2-on-1 verification. If problem in pair, split it, be assigned under random other pair for 2-on-1 verification. Fully anonymous. The proof-of-unique-human is mixed. Untraceable from month to month.

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 {

    uint constant public genesis = 1631944800;
    uint constant public period = 4 weeks;

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

    function hour(uint _t) public pure returns (uint) { return 1 + uint(keccak256(abi.encode(_t)))%24; }
    function pseudonymEvent(uint _t) public pure returns (uint) { return toSeconds(_t) + hour(_t)*1 hours; }

    uint entropy;

    function getRandomNumber() internal returns (uint) { return entropy = uint(keccak256(abi.encodePacked(blockhash(block.number - 1), entropy))); }

    struct Nym { uint id; bool verified; }
    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 registry;
    mapping (uint => mapping (uint => Pair)) public pair;
    mapping (uint => mapping (address => Court)) public court;

    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;

    constructor(address _genesis) { balanceOf[0][Token.Registration][_genesis] = type(uint).max; }

    function registered(uint _t) public view returns (uint) { return registry[_t].length; }
    function pairs(uint _t) public view returns (uint) { return (registered(_t)/2); }
    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 deductToken(uint _t, Token _token) internal { require(balanceOf[_t][_token][msg.sender] >= 1); balanceOf[_t][_token][msg.sender]--; }

    function register() public {
        uint t = schedule();
        deductToken(t, Token.Registration);
        uint id = 1;
        if(registered(t) != 0) {
            id += getRandomNumber() % registered(t);
            registry[t].push(registry[t][id-1]);
            nym[t][registry[t][id-1]].id = registered(t);
        }
        else registry[t].push();
        registry[t][id-1] = msg.sender;
        nym[t][msg.sender].id = id;
    }
    function immigrate() external {
        uint t = schedule();
        deductToken(t, Token.Immigration);
        court[t][msg.sender].id = 1 + getRandomNumber()%type(uint).max;
    }

    function verify() external {
        uint t = schedule()-1;
        require(block.timestamp > pseudonymEvent(t+1));
        uint id = nym[t][msg.sender].id;
        require(id != 0);
        uint _pair = getPair(id);
        require(pair[t][_pair].disputed == false);
        pair[t][_pair].verified[id%2] = true;
    }
    function judge(address _court) external {
        uint t = schedule()-1;
        require(block.timestamp > pseudonymEvent(t+1));
        uint signer = nym[t][msg.sender].id;
        require(getCourt(t, court[t][_court].id) == getPair(signer));
    }
}

```

```

    court[t][_court].verified[signer%2] = true;
}

function _verified(uint _t, uint _pair) internal {
    require(pairVerified(_t, _pair));
    balanceOf[_t+1][Token.Personhood][msg.sender]++;
    balanceOf[_t+1][Token.Registration][msg.sender]++;
    balanceOf[_t+1][Token.Immigration][msg.sender]++;
}

function nymVerified() external {
    uint t = schedule()-1;
    require(nym[t][msg.sender].verified == false);
    _verified(t, getPair(nym[t][msg.sender].id));
    nym[t][msg.sender].verified = true;
}

function courtVerified() external {
    uint t = schedule()-1;
    require(court[t][msg.sender].verified[0] == true && court[t][msg.sender].verified[1] == true);
    _verified(t, getCourt(t, court[t][msg.sender].id));
    delete court[t][msg.sender];
}

function claimPersonhood() external {
    uint t = schedule();
    deductToken(t, Token.Personhood);
    proofOfUniqueHuman[t][msg.sender] = true;
    population[t]++;
}

function dispute() external {
    uint t = schedule()-1;
    uint _pair = getPair(nym[t][msg.sender].id);
    require(_pair != 0);
    require(!pairVerified(t, _pair));
    pair[t][_pair].disputed = true;
}

function _reassign(uint _t, uint _id, uint _pair) internal {
    require(pair[_t][_pair].disputed == true);
    court[_t][msg.sender].id = 1 + uint(keccak256(abi.encodePacked(msg.sender, _id)))%type(uint).max;
}

function reassignNym() external {
    uint t = schedule()-1;
    uint id = nym[t][msg.sender].id;
    _reassign(t, id, getPair(id));
    delete nym[t][msg.sender];
}

function reassignCourt() external {
    uint t = schedule()-1;
    uint id = court[t][msg.sender].id;
    delete court[t][msg.sender].verified;
    _reassign(t, id, getCourt(t, id));
}

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;
}
}

```