BitPeople: A Proof-of-Unique-Human System

Johan Nygren, johanngrn@gmail.com

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 The proofs, valid for a month, can be disposed of once no longer valid, and are untraceable from month to month.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

BitPeople 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 BitPeople. Each pair is concerned only with itself, compartmentalized, operating identically regardless of how many times the population doubles in size.

Pseudonym events, a new population registry

With global and simultaneous pseudonym events, it is possible to prove that every person on Earth only has one proof-of-unique-person within the system. The ideal organization is that people form pairs. Each person is paired with a stranger, and with a new stranger every month, using a new public key that is untraceable to the account used the previous month. During the event, both partners in a pair have to mutually verify one another, using the verify() function. If a person has any reason to not verify the account they are paired with, they can use the dispute() function to break up their pair, and each be assigned to a "court" under another pair.

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

The key to BitPeople 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. 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(bool _early) external {
    uint t = schedule()-(1-boolToUint(_early));
    if(_early == true) require(halftime(t));
    uint id = getPair(nym[t][msg.sender].id);
    require(id != 0);
    if(_early == false) require(!pairVerified(t, id));
    pair[t][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.

Exponential growth from 2 people to 8 billion

The pairs are the control structure of BitPeople, and each pair is concerned only with itself, regardless of how many times the population doubles in size. The population is allowed to roughly double in size each event, made possible by that each person in a pair verified during an event can be authorized to invite another person to the next event. The exact maximum growth rate is $x(n)=x(n-1)+x(n-2)$, i.e. ϕ^n, slightly slower than 2^n growth. This is close to a maximum growth of one immigrant per pair, a socially comfortable maximum rate. This allows the population to grow from 2 people to 10 billion in 50 events. There is no theoretical upper limit to this scalability mechanism, BitPeople has infinite scalability. The constructor allows the contract to intialize 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.Nym][_genesis] = _population;
}
```

The governance in BitPeople is based on the principle that the average person is good. Decision-making is done by allocating authority randomly within the population. The issuance of opt-in permits conform in the logical way to this principle. People can once per month use borderVote() to authorize a randomly selected person to issue an opt-in permit. This lets the population balance the need to invite people with the risk of attack vectors that target the border. The randomization of who gets to issue a permit relies on the random list of registered people generated for the previous event, and therefore excludes people who opt-in or used dispute() and were verified in a "court" (and that is why the maximum growth is $x(n)=x(n-1)+x(n-2)$.)

```
function borderVote(uint _limit) external {
    uint t = schedule();
    deductToken(t, Token.Border);
    if(_limit != 0) {
        require(permits[t] < _limit);
        limit[t][_limit]++;
    }
    if(excess[t] == 0) permits[t]++;
    else excess[t]--;
    excess[t] += limit[t][permits[t]];
}
```

Use the parameter _limit to set an upper limit to how many opt-in permits you authorize, and your vote will only count up to that many permits.

Randomization, Vires in Numeris

The population is randomized by that each person who invokes register() will take the place of a randomly selected person, and move that person to the end. This mechanism keeps the computational cost per person low. The registration is open during the first half of the period (the second half is for "pre-meetings" to prevent the man-in-the-middle-attack), and the shuffle is finished after that when the registration closes. Once shuffle is complete, the first two people in the registry list will form pair 1, the next two pair 2, the next two pair 3, etc. The pair of any account can be calculated with the getPair() function.

```
uint entropy;

function getRandomNumber() internal returns (uint) {
    return entropy = uint(keccak256(abi.encode(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() external {
    uint t = schedule();
    deductToken(t, Token.Nym);
    registry[t].push();
    uint counter = registered(t);
    uint id = 1 + getRandomNumber()%counter;
    registry[t][counter-1] = registry[t][id-1];
    registry[t][id-1] = msg.sender;
    nym[t][registry[t][counter-1]].id = counter;
    nym[t][msg.sender].id = id;
}
```

Collusion attacks

BitPeople is vulnerable to 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., BitPeople is a 66% majority controlled system.
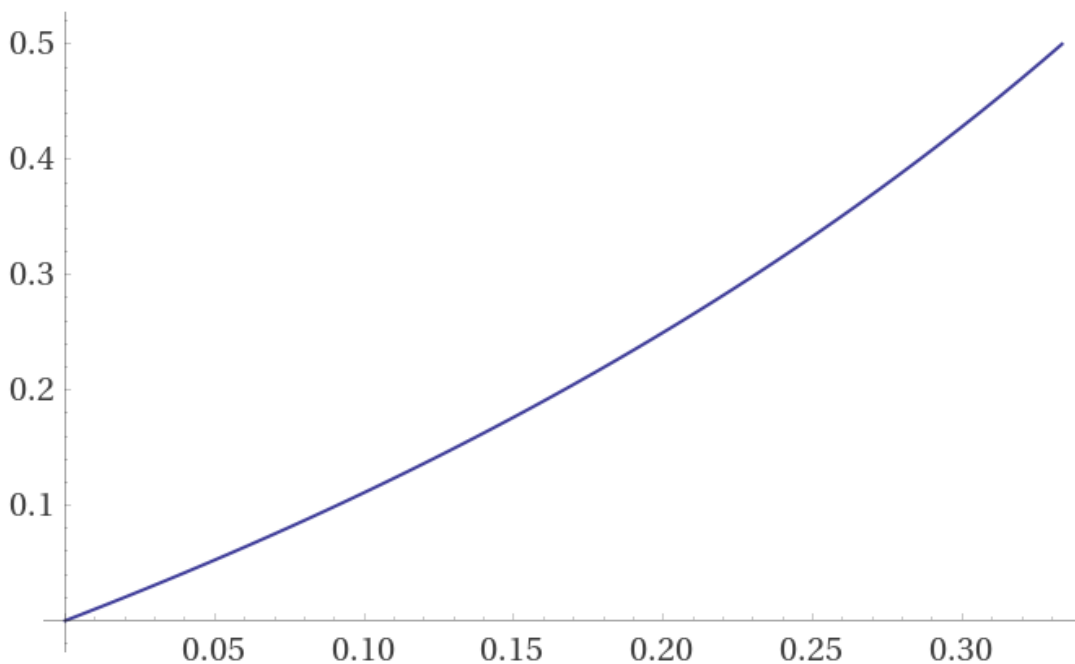
plot (x^2/(1−2x)+x)/(1+x^2/(1−2x)) from 0 to 1/3 | Computed by Wolfram|Alpha

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 = 1654930800;
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 uint(keccak256(abi.encode(_t)))%24; }
function pseudonymEvent(uint _t) public pure returns (uint) { return toSeconds(_t) + hour(_t)*1 hours; }
```

To schedule the event on the weekend for all time zones, Friday 19:00 UTC-12 to Sunday 20:00 UTC+14 is a good time window, achieved with "genesis" set to 07:00 UTC on a Saturday one month before the first event.

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 { ProofOfUniqueHuman, Nym, Permit, Border }

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

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. But video manipulation attacks have to be considered. To have extra security, a "handshake" to secure the channel is added, in a way that is as "Turing safe" (same difficulty for breaking Turing test) as the actual event itself.

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. Before exchanging decryption keys, the pair also meets (otherwise, the mechanism can be attacked.) The pair then decrypts their numbers, and meet at the time that the random number specifies. This "pre-meeting" can only take place if they got the same number, and it proves that an honest message could be exchanged. The public keys that were included in the message are used to secure the channel.

In the source code, half of the month is allocated for the pre-meetings, by adding require(!halftime) to register(). The dispute() function and reassign functions are also accessible during the pre-meeting period, specified by setting the argument _early to true.

```
function halftime(uint _t) public view returns (bool) { return(block.timestamp > toSeconds(_t)+period/2); }
```

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 BitPeople {

    uint constant public genesis = 1664607600;
    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 uint(keccak256(abi.encode(_t)))%24; }
    function pseudonymEvent(uint _t) public pure returns (uint) { return toSeconds(_t) + hour(_t)*1 hours; }

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

    uint entropy;
    function getRandomNumber() internal returns (uint) { return entropy = uint(keccak256(abi.encode(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 { ProofOfUniqueHuman, Nym, Permit, Border }

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

    mapping (uint => uint) permits;
    mapping (uint => mapping (uint => uint)) limit;
    mapping (uint => uint) excess;

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

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

    function boolToUint(bool _bool) internal pure returns (uint) { return _bool ? 1 : 0; }

    function register() external {
        uint t = schedule();
        require(!halftime(t));
        deductToken(t, Token.Nym);
        registry[t].push();
        uint counter = registered(t);
        uint id = 1 + getRandomNumber()%counter;
        registry[t][counter-1] = registry[t][id-1];
        registry[t][id-1] = msg.sender;
        nym[t][registry[t][counter-1]].id = counter;
        nym[t][msg.sender].id = id;
    }
    function optIn() external {
        uint t = schedule();
        deductToken(t, Token.Permit);
        court[t][msg.sender].id = getRandomNumber();
    }

    function verify() external {
        uint t = schedule()-1;
        require(block.timestamp > pseudonymEvent(t+1));
        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 _court) external {
        uint t = schedule()-1;
        require(block.timestamp > pseudonymEvent(t+1));
        uint signer = nym[t][msg.sender].id;
```

```solidity
        require(signer != 0);
        require(getCourt(t, court[t][_court].id) == getPair(signer));
        court[t][_court].verified[signer%2] = true;
    }

    function allocateTokens(uint _t) internal {
        balanceOf[_t][Token.ProofOfUniqueHuman][msg.sender]++;
        balanceOf[_t][Token.Nym][msg.sender]++;
        balanceOf[_t][Token.Border][msg.sender]++;
    }
    function nymVerified() external {
        uint t = schedule()-1;
        require(nym[t][msg.sender].verified == false);
        uint id = nym[t][msg.sender].id;
        require(pairVerified(t, getPair(id)));
        allocateTokens(t+1);
        if(id <= permits[t]) balanceOf[t+1][Token.Permit][msg.sender]++;
        nym[t][msg.sender].verified = true;
    }
    function courtVerified() external {
        uint t = schedule()-1;
        require(pairVerified(t, getCourt(t, court[t][msg.sender].id)));
        require(court[t][msg.sender].verified[0] == true && court[t][msg.sender].verified[1] == true);
        allocateTokens(t+1);
        delete court[t][msg.sender];
    }

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

    function dispute(bool _early) external {
        uint t = schedule()-(1-boolToUint(_early));
        if(_early == true) require(halftime(t));
        uint id = getPair(nym[t][msg.sender].id);
        require(id != 0);
        if(_early == false) require(!pairVerified(t, id));
        pair[t][id].disputed = true;
    }
    function reassignNym(bool _early) external {
        uint t = schedule()-(1-boolToUint(_early));
        uint id = nym[t][msg.sender].id;
        require(pair[t][getPair(id)].disputed == true);
        delete nym[t][msg.sender];
        court[t][msg.sender].id = uint(keccak256(abi.encode(id)));
    }
    function reassignCourt(bool _early) external {
        uint t = schedule()-(1-boolToUint(_early));
        uint id = court[t][msg.sender].id;
        require(pair[t][getCourt(t, id)].disputed == true);
        delete court[t][msg.sender].verified;
        court[t][msg.sender].id = uint(keccak256(abi.encode(id)));
    }

    function borderVote(uint _limit) external {
        uint t = schedule();
        deductToken(t, Token.Border);
        if(_limit != 0) {
            require(permits[t] < _limit);
            limit[t][_limit]++;
        }
        if(excess[t] == 0) permits[t]++;
        else excess[t]--;
        excess[t] += limit[t][permits[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;
    }
}
```