PLANETARY DAO

Technical White Paper

2022-06-15

J. A. Smith§ jsmith_life@proton.me

W3ID - Planet Earth Citizenship v6.1

A Proof-of-Unique-Human System

"The Rights of Citizenship"

Abstract. Planet Earth Citizenship 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.

Introduction

Planetary Citizenship 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 Planetary Citizenship. 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, using a new public key that is untraceable to the account. 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 Planetary Citizenship 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() external {
uint t = schedule()-1;
uint id = getPair(nym[t][msg.sender].id);
require(id != 0);
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.

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

The pairs are the control structure of Planetary Citizenship, 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 in 10 events, 10^3 to 10^6 in 20 events, and 10^6 to 10^9 in 30 events.
There is no theoretical upper limit to this scalability mechanism, Planetary Citizenship has infinite scalability. The constructor allows the contract to be initialized 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 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 entire period, and the shuffle is finished at the end of the period 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 = unit(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() public {
uint t = schedule();
deductToken(t, Token.Registration);
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

Planetary Citizenship 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., Planetary System is a 66% majority controlled System.

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] >=
_
balanceOf[_
t][_
token][_
from] -=
value;
_
balanceOf[_
t][_
token][_
to] +=
value;
_
value);
}
function transfer(address
to, uint
value, Token
_
_
_
token) external {
_
transfer(schedule(), msg.sender, _
to, _
value, _
token);external {
}
function approve(address
_
spender, uint
value, Token
_
allowed[schedule()][_
token][msg.sender][_
spender] =
}
function transferFrom(address
from, address
_
_
to, uint
uint t = schedule();
require(allowed[t][_
token][_
from][msg.sender] >=
_
_
transfer(t, _
from, _
to, _
value, _
token);
allowed[t][_
token][_
from][msg.sender] -=
```

value;

_
}
value);

_
token) external {
value;

_
value, Token

_
_
token)

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]++;
}

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.Public keys are exchanged along with the numbers, as part of the encrypted message. Before
exchanging encryption 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.

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 Planetcity {
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 { 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
type(uint).max; }
_genesis) { balanceOf[0][Token.Registration][_genesis] =
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() public {
uint t = schedule();
require(!halftime(t));
deductToken(t, Token.Registration);
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 immigrate() external {
uint t = schedule();
deductToken(t, Token.Immigration);
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;
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.Personhood][msg.sender]++;
balanceOf[_
t][Token.Registration][msg.sender]++;
balanceOf[_
t][Token.Immigration][msg.sender]++;
}
function nymVerified() external {
uint t = schedule()-1;== true);
require(nym[t][msg.sender].verified == false);
require(pairVerified(t, getPair(nym[t][msg.sender].id)));
allocateTokens(t+1);
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]
allocateTokens(t+1);
delete court[t][msg.sender];
}
function claimPersonhood() external {
uint t = schedule();
deductToken(t, Token.Personhood);
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
internal {
_
transfer(uint
t, address
from, address
to, uint
_
_
_
_
value, Token
require(balanceOf[_
t][_
token][_
from] >=
```

```
_
balanceOf[_
t][_
token][_
from] -=
value;
_
balanceOf[_
t][_
token][_
to] +=
value;
_
value);
}
function transfer(address
to, uint
value, Token
_
_
_
token) external {
_
transfer(schedule(), msg.sender, _
to, _
value, _
token);
}
function approve(address
_
spender, uint
value, Token
_
_
allowed[schedule()][_
token][msg.sender][_
spender] =
_
token) external {
value;
}
function transferFrom(address
from, address
to, uint
_
_
_
uint t = schedule();
require(allowed[t][_
token][_
from][msg.sender] >=
_
_
transfer(t, _
from, _
to, _
value, _
```

```
token);
allowed[t][_
token][_
from][msg.sender] -=
value;
_
}
}
```