1 Background

Below we outline commonly used parameters. The notation makes use of exponential Elgamal

1. $H()$: cryptographic hash function
2. G: generator point
3. *Alice*: a user who at a later point becomes infected.
4. *Bob*: a healthy user who wants to perform set intersection with the uploaded data set.
5. p_i : *Alice*s location data
6. b_k: *Bob*s location data
7. s_i: a set of servers holding encrypted trajectory data
8. $y, y1, y2$: ephemeral secrets

2 Server Setup

Each server s_i generates a private,public key pair r_i, G^{r_i} and server key $Q_{\text{servers}} = G^{\prod_i r_i}$ generated through a multiparty Diffie-Hellman Key Exchange protocol and publishes the public keys:

$$G^{r_i}, Q_{\text{servers}}$$

3 Background Mode and Upload

Alice initializes an ephemeral secret $x \in \mathbb{Z}_p$, and actively in the background collects her location data p_i and proceeds to hash with $H()$ and encrypts the point with the server shared key Q_{servers} as:

$$Q^{xH(p_i)}$$

During the upload phase, Alice selects any number of servers s_i to upload the data:

$$G^x, D = \{Q^{xH(p_i)}, ...\}$$

The set intersection does not require interaction with Alice, thus she may go offline after the upload. To further safeguard Alice, she may dispose of the secret x after storing G^x, Q^x.

4 Query and Compare

We now describe the interactive protocol phases: $Phase_{source \rightarrow destination}$ between Bob and $s_i : i = 2$ to perform set intersection. Let:$y, y1, y2 \in \mathbb{Z}_p$. Bob retrieves $[G^{x_j}, D_j] \forall j$ infected persons from s_i. Let b_k : Bobs location data

Bob generates ephemeral secret y and calculates

$$Phase_{Bob \rightarrow s_1} = G^{xyH(b)} \tag{1}$$

s_1 generates an ephemeral key y_1:

$$\begin{aligned} Phase_{s_1 \rightarrow Bob} &= [Phase_{Bob \rightarrow s_1}^{r_1 + y_1}, H(G^{r_2 y_1})] \\ &= [G^{xyH(b) r_1 + y_1}, H(G^{r_2 y_1})] \end{aligned} \tag{2}$$

Bob relays $Phase_{s_1 \rightarrow Bob}$ to s_2. s_2 generates an ephemeral key y_2 :

$$\begin{aligned} Phase_{s_2 \rightarrow Bob} &= [Phase_{s_1 \rightarrow Bob}^{r_2 + y_2}, H(G^{y_2})] \\ &= [G^{(xyH(b) r_1 + y_1) r_2 + y_2}, H(G^{y_2})] \end{aligned} \tag{3}$$

Bob calculates:

$$\begin{aligned} Result_{Bob} &= Phase_{s_2 \rightarrow Bob} - D^y \\ &= G^{(xyH(b) r_1 + y_1) r_2 + y_2} - G^{xH(p) r_1 r_2 y} \\ &= G^{xyH(b) r_1 r2} + G^{y_1 r_2} + G^{y_2} - G^{xH(p) r_1 r_2 y} \\ &= G^{y_1 r_2} + G^{y_2}, H(b) \iff H(p) \end{aligned} \tag{4}$$

Bob can now check the results of the set intersection against $s_1 || s_2$

$$\begin{aligned} Result_{s_1 \rightarrow Bob} &= H(Result_{Bob} - G^{r_2 y_1}) \\ &= H(G^{y_2}) \end{aligned}$$

or,

$$\begin{aligned} Result_{s_2 \rightarrow Bob} &= H(Result_{Bob} - G^{y_2}) \\ &= H(G^{r_2 y_1}) \end{aligned}$$

The returned result will equal the hash of(2) or (3), H(b) $\iff H(p)$

5 Analysis

The scheme preserves privacy of trajectory data for any permutation of colluding servers s_i and users u_k as long as 1 server remains honest.

//TODO: communication complexity discussion and batching

6 Data Bond

We can further harden collusion resistance by introducing economic incentives. Every Server that participates in the network places a bond in a smart-contract escrow and signs it with its private key r_i using ECDSA.

If s_i leaks its private key: r_i, the entity which obtains the key may claim the funds by signing a destination address with r_i, thus unlocking the funds in the smart contract to an address controlled by the user. More so, given a high enough economic incentive, other servers will prefer claiming the data bond over collusion.

We outline an Ethereum smart-contract implementation extended from Open Zeppelins' Escrow contract.

Each s_i executes a $deposit()$ leveraging their public key G^{r_i}. At anytime, a user may retrieve the escrow balance of any s_i. If a server leaks private key r_i, a *claimer* leverages *EllipticCurveDigitalSignatureScheme* to sign a destination address controlled by *claimer* with the leaked private key r_i and unlocks the funds via $claim()$. The smart contract will further broadcast a *LeakClaimed* event on the Ethereum blockchain.

```
pragma solidity ^0.6.0;

import {ECDSA } from "./cryptography/ECDSA.sol";
import './math/SafeMath.sol';
import './access/Ownable.sol';
import './utils/Address.sol';


contract DataBond is Ownable{

    using SafeMath for uint256;
    using Address for address payable;

    event Deposited(address indexed payee, uint256 weiAmount);
    event Withdrawn(address indexed payee, uint256 weiAmount);
    event LeakClaimed(address indexed leaker, address indexed claimer, uint256 weiAmount);

    mapping(address => uint256) private _deposits;



    function depositsOf(bytes calldata publicKey) public view returns (uint256) {
        address payee = toAddressFromPublicKey(publicKey);
        return _deposits[payee];
    }

    /**
     * @dev Stores the sent amount as credit to be withdrawn.
     * @param publicKey The server publicKey.
     */
    function deposit(bytes calldata publicKey) public virtual payable onlyOwner {
        address payee = toAddressFromPublicKey(publicKey);
        uint256 amount = msg.value;
        _deposits[payee] = _deposits[payee].add(amount);
        emit Deposited(payee, amount);
    }



    /**
     * @dev Withdraw accumulated balance for a payee, forwarding all gas to the
     * recipient.
     *
     * WARNING: Forwarding all gas opens the door to reentrancy vulnerabilities.
     * Make sure you trust the recipient, or are either following the
     * checks-effects-interactions pattern or using {ReentrancyGuard}.
     *
     * @param payee The address whose funds will be withdrawn and transferred to.
     */
    function withdraw(address payable payee) public virtual onlyOwner {
        uint256 payment = _deposits[payee];
```

```
        _deposits[payee] = 0;

        payee.sendValue(payment);

        emit Withdrawn(payee, payment);
    }

    function claim(address payable claimer, bytes calldata signature) public virtual{
        address leaker = ECDSA.recover(ECDSA.toEthSignedMessageHash(toBytes32(claimer)), signature);
        uint256 payment = _deposits[leaker];
        _deposits[leaker] = 0;
        claimer.sendValue(payment);
        emit LeakClaimed(leaker, claimer, payment);
    }

    function toBytes(address a) public pure returns (bytes memory) {
        return abi.encodePacked(a);
    }

    function toBytes32(address a) public pure returns (bytes32){
        return bytes32(uint256(a) << 96);
    }

    function toAddressFromPublicKey(bytes calldata b) public pure returns (address){
        return address(int256(keccak256(b)));
    }
}
```

7 Source Code

Github:Protocol-POC