Decentralized Messaging on the Blockchain

Simon Yuan - yuansi@ethz.ch
Fynn Faber - faberf@ethz.ch
Timofey Derkovsky - dtimofey@ethz.ch
Henry Trinh - trinhhe@ethz.ch
Kevin De Keyser - dekevin@ethz.ch

ETH Zürich, 30.04.2018

All contributors contributed equally to this report

The software code which is part of this report is open source and available at https://github.com/ETHBiots2018/JaysonChain.
This project report was written as part of the spring 2018 course 'Blockchain And the Internet of Things (851-0591-01L)' run by M. Dapp, S. Klauser, and D. Helbing.

Contents

1 Abstract

In this paper we present Fletcher, a decentralised untraceable communication protocol useful for the development of non-transparent blockchain applications with use cases such as messaging or supply chain. Using the Fletcher interface, developers can give users greater control over the visibility of their messages. Furthermore, the need for trust in situations where the context of a message plays a role can be eliminated. Fletcher can be deployed on the Ethereum blockchain for decentralisation, preserving data integrity and incentivising fair validation.

2 Motivation

Today blockchain technology is mainly used for reaching consensus about public data or transactions in a decentralised network. It is often claimed that a blockchain solution cannot provide 'non-transparent' security properties such as confidentiality of data, untraceable transactions or user anonymity and that this 'transparent' nature is precisely the advantage of using a blockchain.

Recently however, non-transparent cryptocurrencies like Monero have begun to challenge this view, proving that unauthorised principles can safely validate confidential data. We hope to convince the reader that there exist non-transparent blockchain applications besides cryptocurrencies, many of which have yet to be discovered. While there does not yet exist a reusable framework for developing such applications that is comparable in scope to smart contracts, we show that a less general messaging framework is sufficient for many new applications.

Here are some examples:

Quality Assurance Alice wants to sell her car to Bob. Additionally, she wants to prove to Bob that the car has a low mileage. However, Bob doesn't trust Alice and will only accept the mileage as valid if it is signed by the manufacturer, who tracks the car via GPS and keeps a record of its mileage. Alice and Bob agree that the Manufacturer shouldn't be aware that the car is being sold to Bob. All three participants don't want third parties to be aware that anything is happening. The mileage information should be kept confidential to third parties as well.

Supply Chain Alice owns a factory known for producing quality products. Alice does not handle the distribution herself and instead the product is sold to a store, where it is sold to Bob. The store owner wants to convince Bob that he is buying a quality product, but Bob does not trust the store. Instead Bob only accepts the product if he has a proof that Alice has produced it. Alice doesn't want to publicly announce the products she has manufactured. Neither Bob nor the store owner want Alice to know that a product is being sold. All three participants don't want third parties to be aware of anything that is happening.

A description of how to solve these problems using the Fletcher interface can be found in the 'Case-Studies' section.

3 Interface

The Fletcher interface provides access to a messaging protocol which guarantees some 'non-transparent' security properties. This protocol depends on the Ethereum blockchain for decentralisation, preserving data integrity and incentivising the fair validation of transactions, however it could also be implemented on a trusted centralised server.

Using the interface, multiple users may communicate through a public ledger on a blockchain. In order to send a message, a user must first put it on the ledger using the `put_data()` function and then provide reading access to the recipient using the `disclose()` function. Users may disclose any message that they have access to, including messages they have not authored (not 'put' themselves). We will refer to this as forwarding. This means that disclosing a message also gives the recipient the ability to forward the message without the original author or the one who disclosed it knowing.

We refer to the user who put the message as it's author. We call the user who disclosed the message the discloser. The user to whom the message is disclosed is the recipient. In the case of forwarding, the author is different from the discloser.

Any time a message is disclosed, both the discloser and the recipient obtain a 'proof' that the message has been disclosed. Both parties may choose to put this proof of disclosure on the ledger using the `put_proof()` function. After a proof has been put on the ledger it can be disclosed in the same way a data message can.

This means that the content of a message may either be raw data (data message) or the proof of the disclosure of some original message (proof message). In the latter case the author of the proof message is either the discloser or the recipient of the original message. Note that the original message in a proof message could be another proof message.

The difference between forwarding a message and sending a proof that the message was received is that in the latter case, the user can prove the identity of the discloser to the recipient of the proof.

Users may receive messages that others have disclosed to them by calling the `receive()` function, which returns a datatype containing the content of the message, the public key of the message discloser, the public key of the message author, the access needed to forward the message and the proof of disclosure. If the content of the disclosed message is a proof of disclosure itself then the message content will contain the public key of the author, the discloser and the recipient of the original message as well as the original message content (which could also be another proof). This recursion can be arbitrarily deep, e.g. it is possible to send a proof of disclosure of a message containing a proof of disclosure of a message etc.

It is also possible to put or disclose messages anonymously. In that case the recipient would not know the author or the discloser respectively.

In summary the interface consists of 4 functions:

`put_data(string PuA, string PrA, string data) returns string`
PuA : This is the users public key. Passing the value 0 will result in an anonymous message.
PrA : This is the users private key. Passing the value 0 will result in an anonymous message.
data : This is the content of the message.
returns : This function returns an access string which can then be passed to `disclose`.

`put_proof(string PuA, string PrA, string proof) returns string`
PuA : This is the users public key. Passing the value 0 will result in an anonymous message.
PrA : This is the users private key. Passing the value 0 will result in an anonymous message.
proof : This is the proof string obtained by calling `disclose`.
returns : This function returns an access string which can then be passed to `disclose`.

`disclose(string PuA, string PrA, string PuB, string access) returns string`
PuA : This is the users public key. Passing the value 0 will result in an anonymous message.
PrA : This is the users private key. Passing the value 0 will result in an anonymous message.
PuB : This is the public key of the user one wants to disclose the message to.
access : This is the access string obtained by calling `put_data`, calling `put_proof` or by receiving a message. The access determines which message is being disclosed.
returns : This function returns a string that proves one has disclosed the message to the receiver. It can be passed to `put_proof`.

`receive(uint256 position, string PrA) returns message`
position : This is the index where the function tries to read.
PrA : This is the users private key.
returns : This function returns a "no message" exception if the message was not a message intended for the user. It returns a "out of bounds exception" if the position is greater than the length of the ledger. Else it will return a message as defined below:
message : This data type has six fields:

discloser : This is the public key of the user who disclosed the message. 0 means the message was disclosed anonymously.
author : This is the public key of the user who put the the message. 0 means the message was put anonymously.
access : This is the string needed to forward the message. It can be passed to the disclose function.
proof : This string contains the proof that the message was disclosed to the user by the discloser as opposed to someone else. It can be put on the ledger with `put_proof`
content : This field can either be a string containing the data or, in the case of the message content being a proof of disclosure of an original message, it is another datatype containing 4 fields:

author : The public key of the user who put the original message. A 0 means the original message was put anonymously.
discloser : The public key of the user who disclosed the original message. A 0 means the the original message was disclosed anonymously.
recipient : The public key of the user who received the original message.
content : This is defined recursively as above.

4 Case-studies

4.1 Interface case-study: Quality Assurance

The problem is the same as described in the motivation section.
The manufacturer, Alice and Bob all have public/private key pairs called PuM, PrM, PuA, PrA, PuB and PrB respectively. The manufacturer and Alice know each others public key as do Alice and Bob. The car has a 'non-removable' barcode on it that proves it is car # 43.

The manufacturer sends Alice her mileage in fixed time intervals. (Alternatively, he could send her the mileage every time she requests it.)
`Manufacturer: mileage_access = put_data(PuM, PrM, "Car␣#43␣has␣mileage␣of␣100␣km")`
`Manufacturer: disclose(PuM, PrM, PuA, mileage_access)`

Alice checks for new messages by calling `receive` multiple times, incrementing the `position` parameter until the function returns a message:
`Alice: new_message = receive(*position*, PrA)`
After receiving the message and checking `new_message.content` as well as `new_message.author` she concludes that this is the message she wants to forward to Bob.
`Alice: disclose(PuA, PrA, PuB, new_message.access)`

Bob checks for new messages and eventually receives a message:
`Bob: new_message = receive(*position*, PrB)`
Bob can check `new_message.author` to see that the message is in fact from the manufacturer. After scanning the bar code on the car and comparing it to `new_message.content` he has learned the true mileage of the car from a trusted source and knows Alice has not lied to him about the mileage.

4.2 Interface case-study: Supply-chain

The problem is the same as described in the motivation section.
Alice, Bob and the store owner all have public/private key pairs called PuA, PrA, PuB, PrB, PuS and PrS respectively. Alice and the store owner know each others public key as do the store owner and Bob. The product has a 'non-removable' barcode on it that proves it is product #43.

```
Alice: mileage_access = put_data(PuA, PrA, "I␣released␣a␣product␣with␣barcode␣#43")
Alice: disclose(PuA, PrA, PuS, mileage_access)
Store Owner: new_message = receive(*position*, PrS)
Store Owner: disclose(PuS, PrS, PuB, new_message.access)
Bob: receive(*position*, PrB)
```

4.3 Interface case-study: Broadcast

Alice wants to disclose a message to multiple people ($\text{Bob}_1, \ldots, \text{Bob}_N$) with public keys (PuB1,..., PuBN).

Naively Alice might just simply disclose the messages to every Bob:

```
access = put_data(PuA, PrA, "Hello␣World")
disclose(PuA, PrA, PuB1, access)
⋮
disclose(PuA, PrA, PuBN, access)
```

However, in this scenario the other Bobs do not know for certain that each of them received the (same) message. One might think that, in order for Alice to prove to the Bobs that they all received the same message she would have to send each Bob N-1 proofs (costing her a total of $\mathcal{O}(N^2)$ proofs).

```
   access = put_data(PuA, PrA, "Hello␣World")
proof_B1 = disclose(PuA, PrA, PuB1, access)
```

```
access_B1 = put_proof(PuA, PrA, proof_B1)
disclose(PuA, PrA, PuB2, access_B1)
disclose(PuA, PrA, PuB3, access_B1)
⋮
disclose(PuA, PrA, PuBN, access_B1)
proof_B2 = disclose(PuA, PrA, PuB2, access)
access_B2 = put_proof(PuA, PrA, proof_B2)
disclose(PuA, PrA, PuB1, access_B2)
disclose(PuA, PrA, PuB3, access_B2)
⋮
disclose(PuA, PrA, PuBN, access_B2)
⋮
⋮
disclose(PuA, PrA, PuBN-1, access_BN)
```

However, this is highly redundant. Using recursive proofs we can disclose a message to $B1$ and then proof to $B2$ only the disclosure of $B1$.

```
access = put_data(PuA, PrA, "Hello World")
forward_proof_B1 = disclose(PuA, PrA, PuB1, access)
forward_access_B1 = put_proof(PuA, PrA, forward_proof_B1)
forward_proof_B2 = disclose(PuA, PrA, PuB2, forward_access_B1)
forward_access_B2 = put_proof(PuA, PrA, forward_proof_B2)
forward_proof_B3 = disclose(PuA, PrA, PuB3, forward_access_B2)
...
forward_access_BN-1 = put_proof(PuA, PrA, forward_proof_BN-2)
proof_BN = disclose(PuA, PrA, PuBN, forward_access_BN-1)
```

Now B_N knows the content of the message that A wanted to disclose to him and he knows that all other Bobs received this message. However B_1 still doesn't know that B_N received the message. So we have to do one final backward pass, costing us $\mathcal{O}(N)$ proofs in total.

```
   access_BN = put_proof(PuA, PrA, proof_BN)
backwards_proof_BN-1 = disclose(PuA, PrA, PuBN-1 access_BN)
backwards_access_BN-1 = put_proof(PuA, PrA, backwards_proof_BN-1)
...
backwards_proof_B1 = disclose(PuA, PrA, PuB1, backwards_access_B2)
```

Currently it is not possible to prove that a message happened in the future, which is why a circular protocol is not implemented (could be supported though, see problem of starvation freedom though).

5 Implementation - Protocol

The interface is implemented using a protocol we call the Fletcher protocol. In order to implement this protocol one needs a place to store messages. This can be any sort of ledger (simply an array or a linked list of strings) but if a ledger on the blockchain is used, it has certain advantageous properties which are discussed in the next section.

Besides the ledger one also needs to decide on which asymmetric encryption algorithm (such as RSA-256) and symmetric encryption algorithm (such as the block cipher AES-256) to use.

First we will present a very minimalist protocol with which we can already do untraceable messaging. With this minimal protocol we explain the concept of nonces and indices.

Then we suggest a second protocol (Fletcher protocol), which additionally allows for knowledge of origin and proof of disclosure.

5.1 Minimal Protocol

In this simple protocol every user can send a message to every other user by encrypting the message with the public key of the sender.

$$E_{PuB}(IND \cdot \text{nonce} \cdot x)$$

where
$x :=$ the message Alice A wants to send to Bob B. The information, that the sender of the message is A, should be contained in x. Otherwise B cannot know that the message was from A.
$PuA :=$ Public key of the sender A.
$PrA :=$ Private key of the sender A.
$PuB :=$ Public key of the receiver B.
$PrB :=$ Private key of the receiver B.
$E_{PuZ}(x) :=$ x encrypted with the public key of Z. If PuZ is zero, we define $E_0(x) = x$, so nothing will be encrypted.
$\cdot :=$ Concatenation operator on two strings. If the content between the concatenation operator is of variable length, then the concatenation operator should also include the size of the string that is located between the current and the next concatenation operator (one needs to be a bit careful with this part, but it can be done).
nonce := is a locally generated random number of a fixed length (for our example we will use a 256-bit unsigned integer). This number should be different in every message.

Lastly the IND is a 256-bit unsigned integer representing the index of where the message will be located on the message table. The implementation of the sender can get this number using `getTableLength()`. Whenever it tries to add something to the message table it passes IND as an `_expectedIndex` to the `addMessage(string _message, uint256 _expectedIndex)` function. That way the sender can be sure that his message will end up at index IND, making it a valid message.

Reading the message - The reader can start from the top of the ledger and decrypt every message on the blockchain one-by-one with his private key. This is done off-chain and doesn't cost any gas. Whenever the decrypted message contains the correct IND at the start (if IND matches the index of the ledger) then this message is assumed to be for the reader (it is very unlikely that a 256-bit number matches IND randomly).

Safety - If we were to only encrypt $E_{PuB}(IND \cdot x)$, people could guess a message $IND \cdot y$ and then encrypt it with every public key on the ledger and compare the message with the message on the blockchain. If $E_{PuB}(IND \cdot y) = E_{PuB}(IND \cdot x)$ it would be very likely that $x = y$. This is solved by introducing a sufficiently long nonce, because then the attacker would also need to guess the nonce, which is computationally infeasible. Additionally adding the IND is important, because otherwise one could just copy a message from the ledger and add it again. A receiver doesn't know who sent the message to him and he cannot trust x (unless it contains some signature by A. This will be part of the Fletcher protocol).

Proof of received message - The sender A can prove to C that he sent a certain message to B. He does this by sending a message with the following content to C: $IND \cdot \text{nonce} \cdot x \cdot PuB$. Now C can encrypt $IND \cdot \text{nonce} \cdot x$ with PuB and check whether this exact same message was located at

the index IND on the ledger. Now C knows that B has got a message with the content x. C can then similarly share this information with D and so on.

5.2 Fletcher Protocol

In this protocol every message between two parties A and B can be shown to have happened by both A and B to a third party. Messages are written to a ledger and only the author can read the messages until he discloses the message to other people.

5.2.1 Putting a data message

`put_data(string PuA, string PrA, string data) returns string`

This is how the implementation would write down a message $x := data$ on the ledger, after the user calls the aforementioned function.

1. **Generate key** Generate a new random symmetric key for the message with the chosen block cipher (such as AES256).

2. **Put data message** Put ones message on the ledger encrypted with this new symmetric key:

$$E_{sym}(\text{MT1} \cdot [DM/PM = 0] \cdot \text{nonce} \cdot PuA \cdot E_{PrA}(IND) \cdot x)$$

   If someone correctly decrypts the message, the left most part will be $MT1$ (a sufficiently long identifier to state that this is a symmetrically encrypted message and is of version 1. It has to be sufficiently long in order for the probability to be low enough that a decryption with the wrong key results in the same string).
   Concretely, our protocol uses: $MT1$:= "54 68 69 73 20 69 73 20 61 20 73 79 6d 6d 65 74 72 69 63 20 6d 65 73 73 61 67 65 3a 20 76 65 72 73 69 6f 6e 20 31 2e 30 20 20 79 6f 75 27 76 65 20 67 6f 74 20 62 61 73 65 64 20 6d 61 69 6c 21"

   The $[DM/PM]$ bit indicates whether this is a *data message* or a *proof message*. Since this is a data message the bit is set to a 0. The index (IND) and nonce are described in the previous protocol, but here the index (IND) is encrypted with the private key of Alice, so Bob additionally knows that the message is from Alice (and doesn't need to trust any information about the sender in x, since the senders public key (PuA) is part of the protocol). If the IND is not correct, i.e. the IND does not match the index, the signature is considered invalid.

3. **Return access** Additionally, the method returns an access to this message which can be disclosed to anyone. An access has the form:

$$\text{sym} \cdot IND$$

   where *sym* was the symmetric key with which the message was encrypted and IND is the index of where it was put on the ledger.

5.2.2 Disclosing a message

`disclose(string PuA, string PrA, string PuB, string access) returns string`

This is how a user (Alice) would disclose a message to a receiver (Bob). Alice needs to pass the access to the direct/proof message as part of the argument to disclose. This method additionally returns a proof, with which Alice can prove to a third party, that she sent the message to Bob.

1. **Disclose** In order to disclose a message to someone else one simply puts the following string on the ledger (arrow / access message). It is untraceable because no other user knows that A and B are communicating (unless they somehow have access to the string).

$$E_{PuB}(\text{AT1} \cdot \text{nonce} \cdot PuA \cdot E_{PrA}(IND) \cdot \text{access})$$

   AT1 is a sufficiently long identifier, which states that this is an asymmetrically encrypted message of version 1.
   Concretely, the protocol uses: AT1 := "54 68 69 73 20 69 73 20 61 20 61 73 79 6d 6d 65 74 72 69 63 20 6d 65 73 73 61 67 65 3a 20 76 65 72 73 69 6f 6e 20 31 2e 30 20 79 6f 75 27 76 65 20 67 6f 74 20 62 61 73 65 64 20 6d 61 69 6c 21".

   PuA is the public key of the discloser. Notice that once a message has been disclosed, the author of the message referenced in access does not need to be the same person as the discloser of the message.

IND is the index of where this new asymmetric access / arrow message will land on the ledger (encrypted by the discloser, so one knows the message is valid).
The entire message is encrypted with the public key of the receiver and we add a nonce so that the message cannot be guessed.

2. **Proof** The proof string consists out of two parts $PuB \cdot \text{ProofMsg}$, where
ProofMsg := the asymmetric message, but not encrypted: $\text{AT1} \cdot \text{nonce} \cdot PuA \cdot E_{PrA}(IND) \cdot \text{access}$
Given these two pieces of information a third party knows that PuA disclosed the message at access to PuB with the message at index IND. The third party can then verify whether this message was actually put on the ledger by encrypting $E_{PuB}(ProofMsg)$ and comparing it to the message at IND. If they match he knows that A disclosed the message at *access* to B.

5.2.3 Receiving a message

`receive(uint256 position, string PrB) returns message`

Whenever the receiver calls this function, the implementation will look-up the message at index position and will try to decrypt this message with his private key.

1. **Check AT1 / Bounds** First, if position exceeds the size of the ledger the function will throw an "out of bounds" exception. If the first part of the decrypted string contains an $AT1$, it then knows that it is a valid disclosure intended for B. The message will be of the form:

$$\text{AT1} \cdot \text{nonce} \cdot PuA \cdot E_{PrA}(IND) \cdot \text{sym} \cdot \text{indexOfOrigMsg}$$

If the message decrypted with PrB does not starting with $AT1$, then the `receive` method will return a "no message" exception.

2. **Reading Discloser** Next it will check who disclosed the message (PuA) and see whether $E_{PrA}(IND)$ actually matches `position`. If not it will return a "no message" exception.

discloser := PuA of this arrow.

The fact that such a message was disclosed to B can also be proven by B (not only by the discloser). The proof message is precisely the same:
proof := $\text{AT1} \cdot \text{nonce} \cdot PuA \cdot E_{PrA}(IND) \cdot \text{access}$

3. **Looking at the disclosed message** Then the actual message at indexOfOrigMsg is retrieved and then the implementation tries to decrypt it with the symmetric key *sym*.

If the decrypted message starts with MT1, then the decryption was successful (elsewise return "no message" exception) and the message will be of the form:

$$\text{MT1} \cdot [DM/PM] \cdot \text{nonce} \cdot PuA_2 \cdot E_{PrA}(IND_2) \cdot x$$

author := PuA_2 of this message. Additionally one needs to check whether the decrypted IND_2 matches indexOfOrigMsg, if not return "no message" exception.

content := x, and depending on the $[DM/PM]$ bit is interpreted simply as a direct message or as a proof of another disclosure.

Not only the author, but also the receiver (everyone with the symmetric key) can forward this message.

Lastly, the fact that such a message was disclosed to B can also be proven by B. This access string is precisely the same string as the access string of the author: *access* := $\text{sym} \cdot IND_2$

5.2.4 Putting a proof message

`put_proof(string PuA, string PrA, string proof) returns string`

Both the sender A and the receiver B of an "arrow" (message for disclosing a message from A) can prove to a third party C that B got access to that message (since both `receive` and `put_data` return a proof).

This proof is sent in the same manner as a data message, except that x := `proof` and that the $[DM/PM]$ bit is set to 1:

$$E_{sym}(\text{MT1} \cdot [DM/PM = 1] \cdot \text{nonce} \cdot PuA \cdot E_{PrA}(IND) \cdot x)$$

Notice that this `put_proof` function also returns an access, which can be disclosed and this disclosure can again be proven to another person and so on. For more details check out the interface.

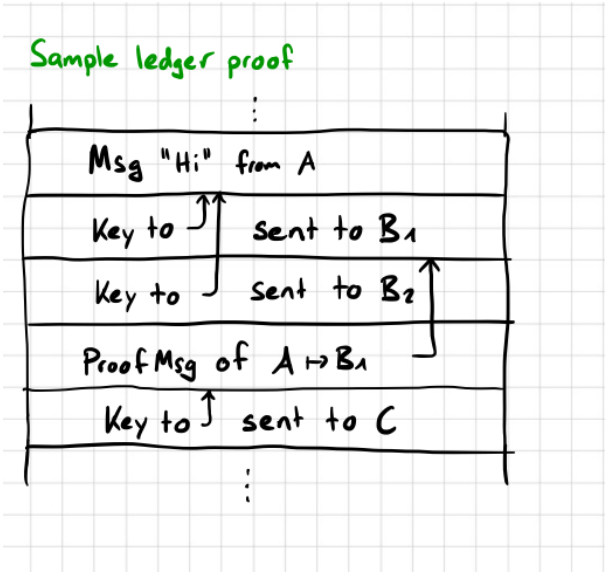

Figure 1: Example message proof on the ledger.
Notice that C doesn't know that A sent the message to B_2, he only knows that he sent it to B_1.
However both A or B_2 could prove to C that their communication happened.

6 Implementation - Ethereum Smart-contract

Reduced to the bare-bones, our messaging system only stores a table of messages (a ledger) for the messages. Here is a Solidity implementation of such a contract:

```
pragma solidity ^0.4.19;
contract MinimalContract {
    struct MessageTableEntry {
        string message;
    }
    MessageTableEntry[] messageTable;

    function addMessage(string _message, uint256 _expectedIndex) public {
        //guarantee that message will be inserted at expected index. Required for cryptographic uses.
        require(messageTable.length == _expectedIndex);
        MessageTableEntry memory entry;
        entry.message = _message;
        messageTable.push(entry);
    }

    function getMessage(uint256 _messageIndex) public view
    returns (string message) {
        require(messageTable.length > _messageIndex);
        MessageTableEntry storage entryPointer = messageTable[_messageIndex];
        return (entryPointer.message);
    }

    function getTableLength() public view returns (uint256 length) { return messageTable.length; }
}
```

6.0.1 Public database of keys

One can extend this smart-contract in various ways, here we added a system where a username can be linked to a public key. Notice that when one publishes their username, everyone can write to them (since the public key is now known). For an in-depth discussion see possible attacks / shortcomings.

```
struct AccountData {
    uint256 publicKey;
}
mapping(string => AccountData) accountDatas;
string[] users;

function publishUsername (string _username, uint256 _publicKey) public {
    //checks if the account was never initialised before
    require(accountDatas[_username].publicKey == 0);
    AccountData memory accountData;
    accountData.publicKey = _publicKey;
    accountDatas[_username] = accountData;
    users.push(_username);
}
//return type requires "pragma experimental ABIEncoderV2;" as of version solidity ^0.4.19
function getUsers() public view returns (string[] userout) { return users; }
function getAccountData(string _user) public view returns (uint256 _publicKey) {
    return accountDatas[_user].publicKey;
}
```

6.1 Advantages of using the blockchain for messaging

Our protocol can be used on any kind of ledger. We could use our protocol on any array, but implementing it with a very simple smart-contract on the blockchain allows for some key advantages:

Traceless reading The `getMessage` function doesn't change the state of the Ethereum blockchain and can therefore be done locally on a full node (without paying ether). Even if one isn't running their own full node, nobody can gain any information if the reader is indiscriminate to all messages on the ledger (For eg. reading all messages as soon as they come up or on every Monday morning).

Immutable messages Since messages can only be added to the ledger, nobody can modify the messages. This makes the messages immutable and independent of the messaging protocol. So while there are cryptographic solutions to detect whether a message has been tampered with (by signing a message) none of this technology needs to be used thanks to the block-chain. Additionally, since everything is stored on the Ethereum smart-contract, the only way the messages can be lost is if all Ethereum nodes and backups somehow get destroyed.

Incentivised to keep the blockchain running People are interested in running the Ethereum blockchain, since that way they can earn gas by mining. Therefore one is not dependent on some kind of central authority that might suddenly stop processing messages.

Spam-protection While reading a message doesn't cost gas, putting a message on the block-chain does. One is therefore heavily incentivised not to spam the system with messages.

Donation of messages The cost of a message is always at the expense of the sender. If the receiver wants to cover the costs and both parties agree, then they could simply send Ether to the sender. This transaction is not anonymous, but there are privacy coins like Monero for anonymous transaction. Ethereum can also simulate similar protocols (ring distributions for example).

6.2 Possible attacks / Shortcomings

A general messaging system can have information leaked in many possible ways. Our system strives to protect the users of the system from as many possible attacks as possible. An attacker can be interested in the following information: The IP address of the sender; the length, the time of the creation or the content of a message; the amount of messages some user has received or sent; to whom a message has been sent to; to whom the message has been forwarded to; who the origin of a forwarded message is and so on.

IP-address One might be able to snoop an IP-address when a messages gets published on the ledger. However given only the IP-address and the message content an attacker cannot gain any information about the receiver and thereby one only knows which IP-address is using the system. One can always side-step this issue by using a system like Tor or even better an incentivised Tor (something similar to Kovri) to hide the IP-Address.

Wallet For each message one can figure out which wallet placed that message. Therefor if one wants to stay anonymous they would need to break any connection from their wallet to their real-life identity (one cannot use simply exchanges for example). This can be done either by only gaining crypto currency through mining or by obfuscating the transaction using mixers (or ring signatures) for example. An advanced blockchain like Monero can even provide zero-knowledge transactions (a similar system can also be achieved on the Ethereum blockchain). Ideally one would even use a different wallet for every message to mitigate damage if one wallet gets linked to the owner in some fashion.

Exchange of public/private keys This can be done off-chain or can be done publicly depending on the use case. For supply chain it might make sense to publish all the public keys on the ledger. Then anyone can write a message to anyone else, but still nobody would know who communicates with whom. However if this is done a wallet/IP-address can get linked to a public key (if these weren't secured as stated above). Spam is not an issue, since sending a message to someone costs gas, but one might still not want to get a message from someone else (although inherently their friends can always give the public key to someone else).

Timing attacks (receiver) As previously mentioned these can only be used if one does not use a full node. Even then if one acts indiscriminately to all messages, no information should get leaked.

Timing attacks (sender) Even if the wallet and IP-address are safe, an ISP might still be able to figure out who a user is by extracting information from their behaviour (timing attacks). This can be a problem even if Tor or a safe VPN is used. To counter this, one can add a random delay before broadcasting the transaction (the message) to the miners.

Content of messages If one stores the messages in plain text, then of course everyone can read the content. The protocols we proposed use symmetric and asymmetric encryption schemes. We discuss certain attacks and bring up solutions (like the nonce) to tackle any attacks on the content of the message.

Length of message The length of a message can usually be obscured by encrypting it. However one might still gain information about whether two people are messaging small texts or say are sending pictures. To obfuscate this one could always make small messages larger or split a large message into various smaller ones, which need to be sent in such a way that one cannot do a timing attack and figure out the real size of the message (one can switch the IP-address for each submessage). Ultimately there will be a trade-off between message length, gas cost and difficulty of hiding the IP-Address.

Large costly messages If the user intends to send large files like images or videos this can become quite expensive. Alternatively they can instead upload the encrypted large files to some decentralised messaging system and only store the hash of the file on the message. However it is not clear how to achieve untraceability on a decentralised messaging system (similar technology to Tor or Kovri).

Latency This is mostly dependent on the current blockchain implementation. The average time required to mine a block on Ethereum is currently around 15-20 seconds, with an additional average 15 seconds to get a message into such a block. Therefor it can have a latency of about 25 seconds for sending a message on the Ethereum blockchain. This all of course depends on the efficiency of the miners and the technology underlying the smart contract. A smaller latency can be practically achieved by paying more gas.

Starvation-freedom If a lot of users are trying to add messages to the ledger, only one person will get a message through with every block. We can solve this with optimistic concurrency. In order to reduce the number of collisions, one can simply create multiple ledger tables on the smart-contract and a sender would then randomly add a message to any of these tables. One only needs to be careful to extend the TS to not only include the index of the ledger, but also which ledger we are talking about.

7 Open questions

Non-forwardable arrows (signatures) What if Alice only wants to send Bob a message, such that Bob knows that the message is from Alice, but he cannot prove to anyone else that this message is from Alice.

Our system currently does not allow this, but this would allow use-cases like the following:

Alice wants to sell her car to Bob. Additionally, she wants to prove to Bob that the car has a low mileage. However, Bob doesn't trust Alice and will only accept the mileage as valid if it is signed by the manufacturer, who tracks the car via GPS and keeps a record of its mileage. The manufacturer can send arbitrary mileage proofs to Alice without Alice being able to forward these proofs to Bob, but as soon as Alice wants a forwardable proof she then has to pay the manufacturer for him to give her such a proof for money.

Promising cryptographic schemes that allow for this might be:
Undeniable signatures
Designated verifier signatures