Trustery: An Identity System Based on Smart Contracts on the Blockchain

FINAL PROJECT REPORT

Mustafa Al-Bassam

KING'S COLLEGE LONDON

April 3, 2016

Abstract

test

Acknowledgements

test

Originality Avowal

I verify that I am the sole author of this report, except for the content of this originality avowal and where explicitly stated to the contrary.

I grant King's College London the right to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to a trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Mustafa Al-Bassam
April 3, 2016

Contents

Chapter 1

Introduction

1.1 Motivation

The motivation for this project to build an identity system based on smart contracts on the blockchain arises from two problems, outlined in this section.

1.1.1 Centralisation and Security Weaknesses of Public Key Infrastructure

The secure operation of SSL/TLS relies on a set of trusted Certificate Authorities (CAs) to authenticate public keys[1]. In practice, the set of trusted CAs are bundled into operating systems and web browsers. Therefore, the Public Key Infrastructure (PKI) is centralized as only CAs chosen by operating system and web browser vendors may issue globally valid certificates.

This system is exclusive; it is expensive and time-consuming to convince operating system and web browser vendors to bundle a CA, therefore entities must usually pay CAs to sign their public keys. For example, it typically takes over 11 months to apply for root CA inclusion in Mozilla products[2].

A major security weakness of this system is that every CA has the ability to issue rogue certificates for any entity. In 2011 the DigiNotar CA issued a rogue certificate for Google which was reported to be used in attempted man-in-the-middle attacks against Google users[3].

PGP is a data encryption and decryption standard that does not use CAs to verify the authenticity of public keys. Instead, it offers a feature that allows individuals to sign other individuals' public keys to certify their authenticity. This creates a web-of-trust model that can navigated to determine the authenticity of public keys belonging to individuals that have no pre-shared secret with each other.[4]

The web-of-trust model is a first step towards a decentralized PKI. However, PGP itself is not a PKI as it does not provide a way to retrieve public keys. Commonly, PKI for PGP is implemented as centralized keyservers (such as pgp.mit.edu) that are used to query for public keys.

Ideally, a PKI for PGP would be be fully decentralized and not rely on centralized servers. Centralized keyservers act as a central point of failure that also allow for censorship, exclusion and alteration of keys by a third party.

1.1.2 Identity

The X.509 standard for certificates on the Internet provides scope for a wide range of identity attributes to be embedded in certificates.[5] Identity attributes include information such as phone number, address and name. This provides a way for certificate authorities to vouch for the identity associations of an organization's online presence.

Adapting this system in a web-of-trust PKI model as described above opens the door for a wide range of identity-related problems to be solved, in contexts where an organization or an individual needs to verify a fact about another organization or individual without trusting paper records that can easily be spoofed unlike cryptographic records.

For example, when an employer needs to verify a potential employee's degree certification, the degree-awarding university can cryptographically sign a degree, and the branch of the government responsible for giving universities degree-awarding powers can cryptographically sign the university's degree-awarding certification. The employer only needs to search the webs-of-trust to trust the branch of the government that is responsible for giving universities degree-awarding powers, and work through the remaining chain-of-trust.

Other examples could be verifying a company's shareholders and directors, verifying the visa of an traveller or verifying the driving license of a citizen.

This not easily possible with the current X.509 certificate standard because the standard does not allow certificate authorities to sign specific and fine-grained attributes in a certificate; certificate authorities must sign the entire certificate or nothing.

1.2 Aims

The aim of this project is to materialize the decentralized public key infrastructure and identity management system as described above.

Specifically, this project will create a system based on a smart contract on the Ethereum blockchain that will enable users to manage identities and attributes associated with identities, including cryptographic keys.

The system will be decentralized and will allow users to sign and verify specific and fine-grained attributes of identities using a web-of-trust model.

Chapter 2

Background review

2.1 Blockchain

The concept of the blockchain was introduced in 2008 by Satoshi Nakamoto in the Bitcoin electronic cash system.[6] Bitcoin is designed as a peer-to-peer network where nodes running the Bitcoin software relay transactions to other nodes. To prevent cash from being double spent, the network reaches a consensus on the ordering of transactions by recording them on the blockchain; the Bitcoin paper describes a process of timestamping transactions by "hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work".

In the Bitcoin network the proof-of-work involves repeatedly hashing blocks of incoming transactions until a hash is found that begins with a certain number of zero bits. This requires CPU power, and so as long as the majority of the CPU power on the network is not controlled by a central authority, a central authority cannot modify the blockchain and reverse transactions. This is because the consensus rule of the network is such that the blockchain with the biggest number of blocks is the correct one, and so an authority that does not have the majority of the network's CPU power is unlikely to outpace the creation of blocks of the rest of the network.

The peer-to-peer nature of the Bitcoin blockchain and the fact that it is designed so that it is computationally expensive (and hence theoretically economically unattractive) for a central authority to take control of the blockchain and reverse transactions makes it a suitable tool to satisfy the decentralized aim of this project.

The idea of this project is therefore to exploit the decentralized data storage and consensus capabilities of the blockchain to store identities and keys for a public key infrastructure, rather than transactions for a cash system.

2.1.1 Smart Contracts and Ethereum

Each Bitcoin transaction references other transactions (called inputs) and creates outputs, which are recorded on the Bitcoin blockchain. The Bitcoin in these outputs can then be "spent" by other transactions. To facilitate the creation of transactions, Bitcoin has a transaction scripting language that is used to specify "locking scripts" for specifying conditions that must be met to spend transaction outputs.[7]

Since the invention of Bitcoin, other forms of blockchain-based systems have emerged that extend the scripting language beyond the purpose of a cash system, to allow for other types of applications to be expressed on the blockchain in the form of "smart contracts".

One such blockchain-based system for smart contracts is Ethereum. The Ethereum white paper describes smart contracts as "complex applications involving having digital assets being directly controlled by a piece of code implementing arbitrary rules".[8]

2.2 Public Key Infrastructure

2.2.1 Certificate Authorities

2.2.2 Web of Trust

2.3 Alternative Identity and Public Key Infrastructure systems

Chapter 3

Requirements & Specification

3.1 Brief

The purpose of this project is to create a system hosted on the Ethereum blockchain and controlled by a smart contract, that allows entities to manage (such as storing, retrieving and verifying in a web-of-trust) identities of itself and other entities.

An entity refers to any participant in the system and may be human or non-human, such as a person, organization or autonomous agent.

An identity is a set of attributes about an entity such as cryptographic keys, names or addresses.

3.2 User Stories

- As a user, I want to publicly publish attributes about myself (my identity) so that other users can act on them.
- As a user, I want to search and retrieve attributes about other users so that I can act on them.
- As a user, I want to sign attributes of other users so that other users are more likely to trust these attributes (for example if the other users also trust me).
- As a user, I want to know which users trust the attributes of other users so that I can decide whether to trust these attributes or not.
- As a user, I want to manage my personal list of trusted identities.

- As a user, I want to publicly publish my PGP key as an attribute of my identity so that other users can encrypt to me.
- As a user, I want to search and retrieve the PGP keys of other users so that I can encrypt to them.

3.3 Functional Requirements

The functional requirements are categorized into two sections:

- The requirements for the smart contract on the blockchain. This represents the rules and the protocol of the system.
- The requirements for the client that interacts with the blockchain using the rules of the smart contract.

3.3.1 Smart Contract

- Entities can publish attributes about themselves (their identity) on the blockchain.
- Data associated with attributes may be stored off the blockchain (for example on IPFS or any arbitrary URI) and linked to from attributes.
- Data associated with attributes that may be stored off the blockchain can have cryptographic hashes representing them published as part of the attributes in the blockchain.
- Entities can sign attributes about entities and publish signatures on the blockchain.
- Entities can revoke their own signatures published on the blockchain.
- Signatures can have optional expiry dates.
- If an attribute is a cryptographic key, an entity can publish cryptographic proof of ownership of the key on the blockchain. Proof of ownership proves that the entity that published the attribute has access to the private keys of a key.

3.3.2 Client

- All of the actions that the smart contract allows can be performed by the client.
- Users can search attributes about identities (entities) from the blockchain.
- Users can retrieve attributes about identities from the blockchain.
- Users can retrieve the signatures associated with an attribute.
- Whether signatures are valid or not should be reflected in the client.
- Users can mark identities as trusted or untrusted in personal trust-stores.
- Users can view a list of the trusted identities in their trust-store.
- Whether an identity is trusted or not is reflected in the client when displaying identities.
- The client can automatically verify the cryptographic proof of ownership of PGP key attributes associated with identities or their attributes.
- Users can publish their PGP key and its associated cryptographic proof of ownership which is automatically generated by the client.
- Where data associated with an attribute is stored off the blockchain, the client can automatically publish attributes with data that is stored using an IPFS or HTTP(S) URI and generate a cryptographic hash associated with the data if necessary.
- Where data associated with an attribute is stored off the blockchain, the client can automatically retrieve data that is stored using an IPFS or HTTP(S) URI and verify the data using its published cryptographic hash if necessary.

3.4 Non-functional Requirements

- The system must be secure. Specifically, this means that:
  - data in the blockchain cannot be modified in an unauthorized manner;

  - the integrity of data and attributes retrieved from the blockchain and off the blockchain must be verified by the client (such as the cryptographic hash of data stored off the blockchain and proof of ownership of cryptographic keys).

- The system should be reliable. All actions of the smart contract can be performed regardless of the state of the system.

- The client should be able to run on most common modern operating systems.

- The financial cost of using the system resulting from blockchain transaction fees should be minimized.

- The system should be scalable to many identities and attributes.

3.5 Specification

The below tables outline the specification for each of the requirements of the smart contract and client, and the non-functional requirements.

3.5.1 Smart Contract

The smart contract is to be written in Solidity, a programming language for writing Ethereum smart contracts.

Requirement	**Specification**
Entities can publish attributes about themselves (their identity) on the blockchain.	Identities are represented by Ethereum addresses. The smart contract will have an *add attribute* transaction which will generate an *add attribute* event with a unique attribute ID.
Data associated with attributes may be stored off the blockchain (for example on IPFS or any arbitrary URI) and linked to from attributes.	The *add attribute* transaction will have a parameter for specifying data, that supports linking to data that is stored off the blockchain.

Data associated with attributes that may be stored off the blockchain can have cryptographic hashes representing them published as part of the attributes in the blockchain.	The *add attribute* transaction will have a parameter for specifying a SHA256 hash.
Entities can sign attributes about entities and publish signatures on the blockchain.	The smart contract will have a *sign attribute* transaction-which will generate a *sign attribute* event with a unique signature ID.
Entities can revoke their own signatures published on the blockchain.	The smart contract will have a *revoke signature* transaction which will generate an *revoke signature* event with a unique revocation ID.
Signatures can have optional expiry dates.	The *sign attribute* transaction will have a parameter to specify an expiry date.
If an attribute is a cryptographic key, an entity can publish cryptographic proof of ownership of the key on the blockchain. Proof of ownership proves that the entity that published the attribute has access to the private keys of a key.	The *add attribute* transaction will have a boolean parameter to flag that a cryptographic proof is attached to the data of the attribute.

3.5.2 Client

The client is to be implemented as a command-line text-based console application written in Python.

Requirement	**Specification**

All of the actions that the smart contract allows can be performed by the client.	There will be console commands for adding attributes, signing attributes and revoking signatures. The commands will take input from the user and form the parameters for each transaction, and communicate with the Ethereum JSON RPC to send the transactions.
Users can search attributes about identities (entities) from the blockchain.	There will be a console command for searching attributes. The command will take input from the user about the parameters to filter the attributes by, and communicate with the Ethereum JSON RPC to filter event logs of added attributes.
Users can retrieve attributes about identities from the blockchain.	There will be a console command for retrieving attributes based on the attribute's ID, which will communicate with the Ethereum JSON RPC to filter a single attribute by its ID.
Users can retrieve the signatures associated with an attribute.	The command for retrieving attributes will also retrieve and display the attribute's signatures by filtering signatures with the attribute's ID using the Ethereum JSON RPC.
Whether signatures are valid or not should be reflected in the client.	The client will check the expiry date associated with each signature and search the event logs for revocations associated with each signature using the Ethereum JSON RPC.

Users can mark identities as trusted or untrusted in personal trust-stores.	There will be a console command to add and remove Ethereum addresses from the list of trusted identities, which will be stored in a local file.
Users can view a list of the trusted identities in their trust-store.	There will be a console command that displays the list of trusted identities.
Whether an identity is trusted or not is reflected in the client when displaying identities.	Each Ethereum address displayed by the various console commands will have an indicator next to it when it is found in the trust-store.
The client can automatically verify the cryptographic proof of ownership of PGP key attributes associated with identities or their attributes.	When a PGP attribute is retrieved, the client will invoke the local GPG binary to verify the signature associated with the attribute data.
Users can publish their PGP key and its associated cryptographic proof of ownership which is automatically generated by the client.	There will be a console command to publish PGP key attributes that invokes the local GPG binary to retrieve the PGP keys and generate signatures for cryptographic proofs of ownership.
Where data associated with an attribute is stored off the blockchain, the client can automatically publish attributes with data that is stored using an IPFS or HTTP(S) URI and generate a cryptographic hash associated with the data if necessary.	The add attribute console command will detect when a HTTP(S) URI is used as input for the data field of an attribute, and automatically download the data over HTTP and generate a SHA256 hash for the data. For IPFS URIs it is not necessary to generate a hash as IPFS URIs include a hash.

Where data associated with an attribute is stored off the blockchain, the client can automatically retrieve data that is stored using an IPFS or HTTP(S) URI and verify the data using its published cryptographic hash if necessary.	The retrieve attribute console command will detect when an IPFS or HTTP(S) URI populates the data field of an attribute, automatically download the data over IPFS or HTTP and verify the SHA256 hash associated with the data in the case of HTTP(S) URIs.

3.5.3 Non-functional Requirements

Requirement	**Specification**
Security: Data in the blockchain cannot be modified in an unauthorized manner.	This is enforced by the Ethereum protocol itself as miners validate all transactions before being included in blocks.
Security: The integrity of data and attributes retrieved from the blockchain and off the blockchain must be verified by the client (such as the cryptographic hash of data stored off the blockchain and proof of ownership of cryptographic keys).	The client will check that the SHA256 hash of data stored off the blockchain matches with the published hash. When a PGP attribute is retrieved, the client will invoke the local GPG binary to verify the signature associated with the attribute data.
The system should be reliable. All actions of the smart contract can be performed regardless of the state of the system.	The smart contract will be implemented so that each attribute is stored independently of each other. This is also partly a condition of the Ethereum protocol itself as each smart contract has a state that is independent of other smart contracts.
The client should be able to run on most common modern operating systems.	The client will be implemented in Python, which is a cross-platform language.

The financial cost of using the system resulting from blockchain transaction fees should be minimized.	The smart contract will be implemented in a way that uses the least operations and stores the least data possible on the blockchain so that the gas price of each transaction is minimized.
The system should be scalable to many identities and attributes.	The Ethereum JSON RPC allows for events to be filtered by specified indexed parameters such attribute ID, attribute owner, etc.

Chapter 4

Design

4.1 Smart contract

The smart contract is the interface on the blockchain that sets the rules for the management and storage of identity attributes and keys, that users using the system must abide by.

When designing a smart contract, it is important to appreciate all the small details such as the way data is stored and the data types used. This is for several reasons:

- The smart contract constitutes the core protocol, functionality and limitations of the whole system.
- Once a smart contract is published and used, its code cannot be change. A lapse in judgement in the design of the smart contract can therefore be fatal.
- Unlike traditional software, every computational operation and line of code in the smart contract contributes greatly to the financial cost of using the system in the form of transaction gas costs.[8]

Designing a smart contract therefore cannot be compared to designing traditional database structures; smart contract design is significantly more intricate. Whereas traditional databases structures can trivially be iterated on without disruption to a system, this is not possible with smart contracts.

For these reasons, it is a good idea to keep the smart contract as minimalistic as possible and to offload as much functionality as possible to the client off the blockchain, in order to reduce the surface for failure and the transaction costs of the system.

4.1.1 Language

Code for Ethereum smart contracts is written in Ethereum Virtual Machine (EVM) code, which is a low-level stack-based bytecode language.[8]

To facilitate the creation of Ethereum smart contracts, several high-level languages for Ethereum exist that are compiled down to EVM code. Two of the main ones are:

- Solidity, a statically typed language with syntax similar to JavaScript.[9]
- Serpent, a dynamically typed language with syntax similar to Python.[10]

Although the two languages are similar in that they are both high-level languages that compile down to EVM code, Solidity was chosen as the smart contract language for this project because it is statically typed.

The fact that Solidity is statically typed means that smart contract code published on the blockchain is more likely to be bug-free as type constraints are enforced at compilation time instead of runtime.

Statically typed smart contracts also allow for explicit control of sizes of data specified to contract methods and stored on the blockchain, which is useful for this particular project as the smart contract is intended for arbitrary types of data to specified for certain parameters and predictable types of data for other parameters.

For example, the Serpent compiler command `serpent mk_signature` shows that the Serpent compiler automatically sets variables to be of type `uint256` (integers of 256 bits) unless other explicitly stated. This would be inefficient for the storage of the cryptographic proof boolean flag specified in the project's smart contract specification, as the variable would be padded to 256 bits[11], therefore increasing the gas price of certain transactions in the system. One of the project's non-functional requirements is to minimize the financial cost of the system.

It would therefore be advantageous to use a language where all types must be explicitly specified, because once a smart contract is published on the blockchain its code cannot be modified.

4.1.2 Entities

Figure 4.1 shows the data entities and their constraints to be defined and manipulated in the smart contract. Solidity allows for the definition of *structs*, a way to define data structures.[12]

Entity entity

The *entity* entity is not explicitly defined as a data structure in the smart

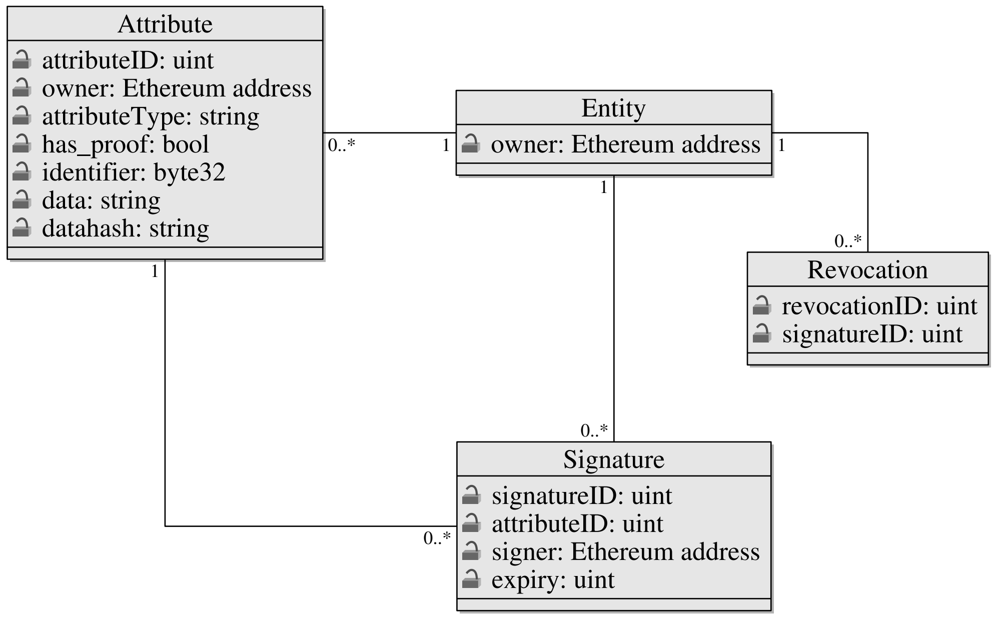

Figure 4.1: Entity-relationship diagram for the Trustery smart contract.

contract. Instead, it is implicit to the implementation of Ethereum.

Each Ethereum transaction has an owner, which is a 20-byte *address* representing a cryptographic public key.[8] For a transaction to be valid, its inputs must be signed using the private key associated with the owner's address.

This makes an Ethereum address a good way to represent an *entity* or a user of the system, as only those who have knowledge of an Ethereum address's private key are able to make transactions on behalf of that address such as adding and signing attributes.

Attributes, signatures and revocations all have one owner entity. For revocations, this is implicit because signature revocations can only be made by the signers of the signatures. For signatures, the owner is referred to as the *signer*.

Attribute entity

The attribute entity represents an attribute about an entity in the system. Its variables are as follows:

- **attributeID**, an unsigned integer that gives each attribute a unique identifier.

- **owner**, an Ethereum address representing the owner of the attribute.

- **attributeType**, a string representing the type of the attribute (such as PGP key or email address). This is a string of dynamic size to give users of the system the flexibility to define their own attribute types of varying lengths, while being efficient as strings are padded to multiples of 32 bytes[11].

- **has_proof**, a boolean to dictate to the client whether the attribute has a cryptographic proof and so can be considered an authentic attribute of an entity after the proof is verified. If the client understands how to verify the cryptographic proof for the attribute type, this enables the client to authenticate the attribute without relying on signatures in the web-of-trust.

  The reason for having this flag instead of allowing the client to simply check the attribute data for a proof is that if the attribute data is stored off the blockchain, it allows the client to know if the attribute has a cryptographic proof without downloading any extra data. If the attribute does not have a cryptographic proof this will save wasted time in downloading data only to find that there is no proof.

- **identifier**, 32 bytes for specifying a searchable identifier for the data in the attribute. This is a 32 byte variable because indexed event parameters (more on that in the next subsection) in Ethereum over 32 bytes are hashed, and some Ethereum implementations do not currently correctly process indexed parameters over 32 bytes. Regardless, 32 bytes is sufficient for an identifier (for example, PGP fingerprints are 160 bits[13]), and if this needs to be increased in the future long identifiers can be hashed client-side.

- **data**, a string representing the actual data of the attribute. This is dynamic because the size of the data can vary greatly depending on the type of the attribute. This field can also be used to specify a off-blockchain URI to the location of the data.

- **datahash**, a string representing a hash of the data of the attribute. This is optional and is intended to be used in cases where data is stored off the blockchain so that the integrity of the data can be verified. Although this is specified in this project as a SHA256 hash, this field is dynamic so that in the future the hashing algorithm can be changed without having to change the smart contract in case a major weakness in SHA256 is found.

Signature entity
This represents a signature of attribute with attribute ID **attributeID** by an Ethereum address **signer**. Each signature has its own unique ID **signatureID**.

There is also an **expiry** field which is an unsigned integer representing the Epoch time when the signature should no longer be considered valid after. Epoch time was chosen as it is a timezone-insensitive way of defining a date.

Revocation entity
This represents a revocation of a signature with signature ID **signatureID**. Each revocation has its own revocation ID **revocationID**.

As signatures can only be revoked by the entity that signed them, the Ethereum address that made the revocation can be easily be inferred from the signatureID.

Note that signatures can have multiple revocations, even though that revoking a signature multiple times is redundant. This is because it would not be gas-efficient to add a condition in the smart contract to check if a signature has already been revoked, and as there is no harm in revoking a signature multiple times the client can ignore subsequent revocations.

4.1.3 Methods and Events

Ethereum smart contracts can have specified methods that can be executed to change the state of the data stored by the smart contract. Method parameters are passed in transactions signed by Ethereum addresses.

In addition to methods, Ethereum has an event logging system. Ethereum methods can emit events to signify a change in the system. Events have parameters and indexed parameters. Ethereum provides APIs to make it easy to search and filter through events by specifying values for indexed parameters.

Ethereum events can only have a maximum of three indexed parameters, called **topics**, so it is important to choose them wisely for each event.[11]

Adding an attribute
The **addAttribute** method enables users to add new attributes to their identities. The method's parameters is all of the fields specified the **attribute** entity specified above, apart from the **owner** field and the **attributeID** field.

There is no need to include the **owner** field as a parameter because as all Ethereum method calls are transactions that are owned by Ethereum addresses, the address that is calling the method can be accessed directly from the smart contract without it being passed by the user as a method parameter.

The **attributeID** field is generated by the method by incrementing the last **attributeID** by one.

This method should emit an **AttributeAdded** event to signify that a new attribute has been added to the system. The parameters for the event should be the same as all the fields for the **attribute** entity, to allow the client to easily retrieve all the information about an attribute.

The event should have three indexed parameters:

- **attributeID**, because it is important to be able to reference and therefore filter an attribute by its unique ID.

- **owner**, to enable users to easily lookup a single entity or user.

- **identifer**, because the purpose of this field is to provide a unique attribute-type-specific reference to an attribute.

Signing an attribute

The **signAttribute** method enables users to sign attributes using their identities. The method's parameters is all of the fields specified the **signature** entity specified above, apart from the **signer** field (for the same reasons specified above for **addAttribute**'s **owner** field) and the **signatureID** field which is generated automatically by the method by incrementation.

The method should emit an **AttributeSigned** event to signify that an attribute has been signed. The parameters for the event should be the same as all the fields for the **signature** entity. As there are only four event parameters, all of them can be indexed expect the **expiry** parameter as it is not very useful to filter signatures by the exact second that they expire.

Revoking a signature

The **revokeSignature** method enables users to revoke signatures that they have made. The method's parameters is all of the fields specified the **revocation** entity specified above, apart from the **revocationID** field which is generated automatically by the method by incrementation.

The method should emit an **SignatureRevoked** event to signify that a signature has been revoked. The parameters for the event should be the same as all the fields for the **revocation** entity. As there are only two event parameters and both of them are IDs, they can both be indexed.

It is important that this method only allows a user to revoke signatures that he or she had made. This should be done by checking that the transaction's owner and the signature signer in reference match, like as follows:

Algorithm 1 Revocation procedure

```
procedure revokeSignature
    if transaction.sender = signatures[signatureID].signer then
        revocationID ← revocations.length + 1
        revocation ← revocations[revocationID]
        revocation.signatureID ← signatureID
        SignatureRevoked(revocationID, signatureID)
        return successful
    else
        return failed
    end if
end procedure
```

4.2 Client

Below is a diagram that outlines the architecture for the client-side portion of the project.

This section will go into the details of the individual components of the architecture.

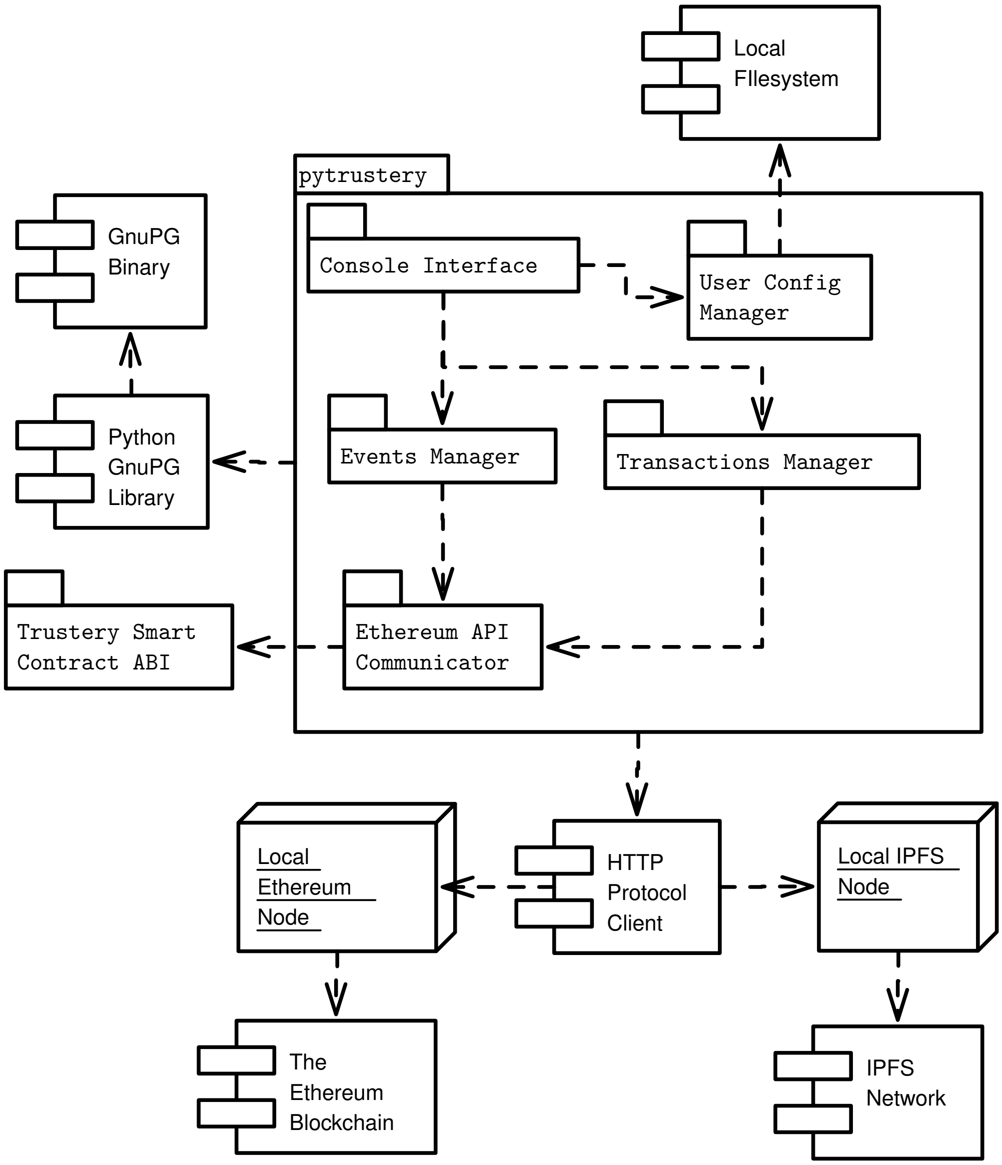

Figure 4.2: Architecture diagram for the Trustery client.

4.2.1 pytrustery package

pytrustery is the name of the Python package for the Trustery client. This is the center of the system, and is responsible for interacting with all the required APIs and external systems required to fulfil the functional requirements of the client.

Language

Several programming languages were considered for the implementation of the command-line client.

It was decided that a high-level interpreted language was more suitable for the client than a low-level compiled language for the following reasons:

- There is a large amount of user input and content processed by the client, so it would be more secure to use a high-level memory safe language to prevent attacks such as buffer overflow. This is especially essential for this project as a security vulnerability in the client could mean identity theft.
- There is no low-level manipulation of hardware needed.
- There is no need for high-speed computation or calculations; all cryptography is done by external libraries.

There are several high-level languages that are suitable for creating command-line applications, such as Python and Ruby.

Although many languages are suitable, Python was chosen over other languages like Ruby for the client because there is a reference implementation of Ethereum in Python (**pyethereum**).[14] This means that modules from the reference implementation can be imported into pytrustery in order to achieve tasks that require making use of the Ethereum Application Binary Interface (ABI)[11] - the interface for interpreting and producing smart contract data.

The **pyethereum** library has modules available for interacting with the Ethereum ABI. This makes creating transactions and interpreting event data from the Ethereum blockchain significantly easier.

4.2.2 Console

4.2.3 Events and Transactions Managers

The

4.2.4 Ethereum API Communicator

4.2.5 User Configuration Manager

4.2.6 IPFS Interaction

4.2.7 PGP Interaction

Chapter 5

Implementation

Chapter 6

Evaluation

Chapter 7

Conclusion

Bibliography

[1] T. Dierks and E. Rescorla. (2008) Rfc 5246 - the transport layer security (tls) protocol version 1.2. [Online]. Available: https://tools.ietf.org/html/rfc5246

[2] (2016) Ca:how to apply. [Online]. Available: https://wiki.mozilla.org/CA:How_to_apply

[3] H. Adkins. (2011) Google online security blog: An update on attempted man-in-the-middle attacks. [Online]. Available: https://googleonlinesecurity.blogspot.fr/2011/08/update-on-attempted-man-in-middle.html

[4] S. Garfinkel, *PGP: Pretty Good Privacy.* OReilly & Associates, 1994.

[5] R. Housley, W. Ford, W. Polk, and D. Solo. (1999) Internet x.509 public key infrastructure certificate and crl profile. [Online]. Available: https://www.ietf.org/rfc/rfc2459

[6] S. Nakamoto. (2008) Bitcoin: A peer-to-peer electronic cash system. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[7] A. M. Antonopoulos, *Mastering Bitcoin: Unlocking Digital Cryptocurrencies.* O'Reilly Media Incorporated, 2014.

[8] V. Buterin *et al.* (2014) A next-generation smart contract and decentralized application platform. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[9] (2015) Solidity 0.2.0 documentation. [Online]. Available: https://solidity.readthedocs.org/en/

[10] (2015) Serpent. [Online]. Available: https://github.com/ethereum/wiki/wiki/Serpent

[11] (2016) Ethereum contract abi. [Online]. Available: https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI

[12] (2015) Types - solidity 0.2.0 documentation. [Online]. Available: https://solidity.readthedocs.org/en/latest/types.html

[13] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. (2007) Rfc 4880 - openpgp message format. [Online]. Available: https://tools.ietf.org/html/rfc4880

[14] (2014) ethereum/pyethereum. [Online]. Available: https://github.com/ethereum/pyethereum