Trustery: A Public Key Infrastructure and Identity System Based on Smart Contracts on the Blockchain

Final Project Report

Mustafa Al-Bassam

King's College London

April 23, 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 23, 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.

Most importantly, having an unforgeable decentralized ledger of keys would provide the transparency needed to quickly identify rogue certificates such as the one issued by DigiNotar CA, as all new certificates would be immediately visible to the network.

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 realise the decentralized public key infrastructure and identity management system that solves the problems 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]

The Ethereum white paper specifies several potential applications for the use of smart contracts such as:

- Token systems representing assets such as company stocks or gold.
- Decentralized marketplaces.
- Decentralized file storage.

One of the other potential applications discussed in the white paper relating to this project is identity and reputation systems. For example, a smart contract can be created for mapping domain names to IP addresses to provide a decentralized domain name registration system. Namecoin is an example of such a system that uses a Bitcoin-like blockchain.[1]

Smart contracts in Ethereum are written in a low-level stack-based bytecode language executed by an Ethereum virtual machine and referred to as Ethereum virtual machine (EVM) code. Smart contracts can also be written in a high-level language such as Serpent and then compiled to EVM code.[8]

The primary proposition of this project is to write such a smart contract with functionality for the operation of a public key infrastructure and identity management system, where public keys and identity attributes are stored on the blockchain and can be managed by the smart contract.

[1] http://namecoin.org/

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.

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

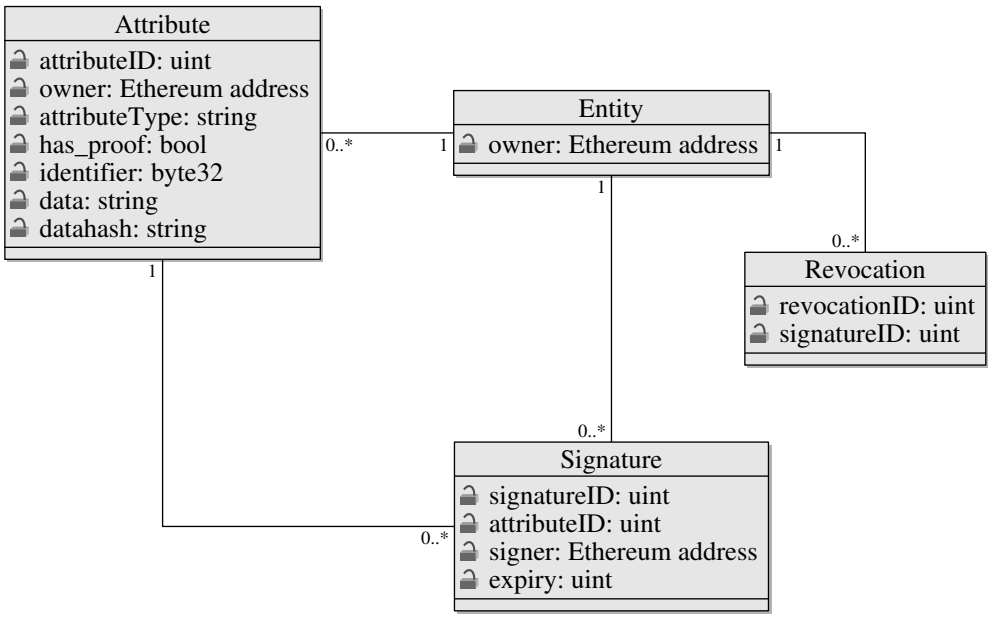

Figure 4.1: Entity-relationship diagram for the Trustery smart contract.

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.

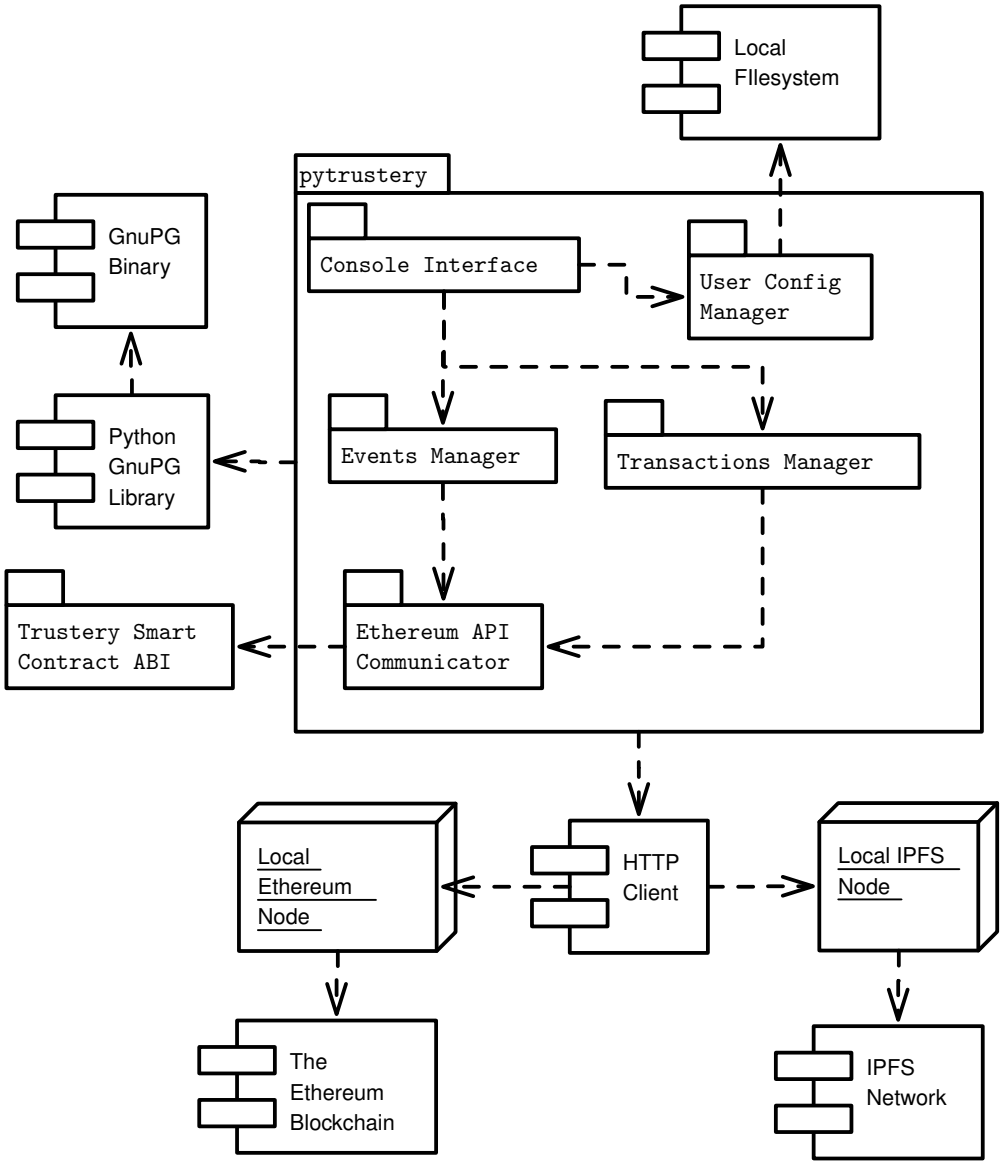

Figure 4.2: Architecture diagram for the Trustery client.

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

The console module can be executed by users to interact with the Trustery system using the command-line.

A Python library called **Click** will be used for the command-line interface. Click is a command-line interface creation kit that aims to make the process of writing command-line interfaces quick.[15]

Using Click, the command-line interface will consist of a set of commands with their own options that can be executed by the user.

The interface will have the following commands to fulfil the functional requirements of the client:

- **add** - adds an attribute.

  Options: attribute type, attribute identifier and attribute data.

- **ipfsadd** - adds an attribute over IPFS.

  Options: attribute type, attribute identifier and attribute data.

- **sign** - signs an attribute.

  Options: attribute ID and expiry time.

- **revoke** - revokes an attribute.

  Options: attribute ID.

- **search** - searches for and filters attributes.

  Options: attribute type, identifier and owner.

- **retrieve** - retrieves information about an attribute and its signatures.

  Options: attribute ID.

- **addpgp** - adds your PGP key as an attribute to your identity.

  Options: PGP key identifier.

- **trust** - adds an identity to your trust store.

  Options: Ethereum address.

- **untrust** - removes an identity from your trust store.

  Options: Ethereum address.

- **trusted** - view the identities in your trust store.

  Options: *none*.

The commands should be executed from the command-line by running **trustery *command***, with the options being specified after the command name.

4.2.3 Events and Transactions Managers

The **events manager** is a singleton class that is responsible for retrieving and filtering events from Ethereum. It has methods for filtering attribute, signature and revocation events by their indexed parameters (as described in *section 4.1.3*).

The **transactions manager** is a singleton class that is responsible for sending transactions using Ethereum. It has methods for fulfilling the methods implemented in the smart contract, including adding attributes, signing attributes and revoking signatures.

The reason for segmenting the two classes by receiving data and sending data from the Ethereum blockchain is because those two functions require inherently different initialization parameters. The events manager is only initialized by the address of the smart contract, and the transactions manager is initialized by the address of the smart contract and also an address belonging to the user.

This means the transactions manager requires the client machine to have access to private keys for signing, whereas the events manager is read-only and so does not need access to any private keys.

This is a useful segregation because it enables a client to be a read-only client if the user only wishes to retrieve attribute data to the blockchain but not publish any actual data.

Furthermore, this also means that the user does not need to unlock their private key and enter their passphrase if the user is only executing read-only commands from the client such as searching for attributes, as read-only commands will only call the events manager class and not the transactions manager class.

Singleton design pattern

A singleton design pattern was chosen for the classes because each method that interacts with Ethereum requires access to the Trustery **contract translator** (from the **pyethereum package**), which provides a programmatic interface to the smart contract ABI and requires initialization. It would be more efficient to initialize the contract translator once for all the methods, as some commands may require multiple calls to methods that interact with Ethereum.

Alternative design

Alternatively, it would be possible to take a full object-orientated approach and abstract attributes, transactions and events as classes.

This level of abstraction is not necessary due to the simplistic design of the console application. Each console command execution is isolated and starts the client process and exits on completion of the command. In addition, the client is only intended to be used by a single user.

These facts combined mean that there is not a large number of attributes, transactions, events and objects with complicated relationships that the client needs to handle, and therefore a high level of object abstraction is not required.

4.2.4 Ethereum API Communicator

The **ethapi** module is responsible for managing tasks relating to direct interaction with the Ethereum system.

The module will initiate a working connection with Ethereum's JSON RPC that can be used by the events and transactions managers. To achieve this, the **ethereum-rpc-client** Python package will be used as the HTTP client for the Ethereum JSON RPC server. The package implements a handful of methods exposed by the Ethereum JSON RPC server.[16]

This module is also responsible for keeping information about the smart contract itself for usage by the events and transactions managers, such as the smart contract ABI to initialize the contract translator with.

Finally, this module will also provide functions to deal with any encoding of data required before sending data off to the Ethereum JSON RPC server. This is intended to standardize the process of using the Ethereum JSON RPC by encoding all the data in a standard way, to prevent bugs caused by data mishandling.

4.2.5 User Configuration Manager

The **userconfig** module is responsible for managing any user state information, including the trust store.

Due to the low volume of user data and lack of need for concurrency as the client serves a single user, it was decided that storing data using flat files is a suitable way to store user state information. The core data of the system regarding the attributes themselves is stored in the blockchain.

ConfigObj is a Python library for reading and writing configuration files using the *INI* file format. This library will be used for managing the user configuration, and was chosen for its simplicity and features.

It supports nested sections which makes it possible to structure configuration sections with a hierarchy, and it comes with a validation system which makes it easy to automatically validate data being stored in the configuration.[17]

Trust store

The user trust store that stores trusted identities will also use this configuration manager to store trust data.

The configuration will have a **truststore** section to represent the trust store. Each Ethereum address trusted by a user will have its own variable in the section, set to the boolean value *true*. Removing an Ethereum address from the trust store is as simple as deleting the variable from the configuration.

Using this design, adding and removing identities from the trust store is are operations that both take *O(1)* time, as adding and removing variables require no iterative computation.

4.2.6 IPFS Interaction

The **ipfs-api** Python package provides IPFS API bindings for Python[18]. Using this package, pytrustery can easily upload and retrieve data to the IPFS network.

The package acts as a HTTP client to communicate with the local IPFS node using its HTTP API.

4.2.7 PGP Interaction

GNU Privacy Guard (GPG) is a complete and free implementation of the OpenPGP standard, that users can install on their local machines.[19]

The **python-gnupg** Python package is a wrapper for the GNU Privacy Guard binary.[20] Using this package, pytrustery can easily retrieve PGP public keys and generate signatures for cryptographic proofs to add as attributes to the blockchain. It can also verify signatures to verify the cryptographic proofs of PGP attributes on the blockchain.

Generating a cryptographic proof for a PGP public key will consist of a signature of the Ethereum address that the PGP key attribute will be published from. This cryptographically proves that the PGP key owner has verified that they want to be associated with the Ethereum address in question.

This binding of PGP key to Ethereum address is useful because it allows users that trust an individual's PGP key to also trust their Ethereum identity.

Chapter 5

Implementation

5.0.8 Methodology

The project's software was implemented in the space of several months using an iterative, agile software development methodology. This worked as the modular nature of the software's design allowed each component to be developed separately. The simplest console commands were implemented first and with each iteration more complicated console commands that required more advanced component interactions were implemented.

The smart contract was implemented first, with the events and transactions managers then being built on top of the smart contract, adding more features that rely on other components such as PGP and IPFS with each iteration. Each iteration relied on the previous iterations. A general overview of the iterations is as follows:

- Iteration 1: code for the smart contract.
- Iteration 2: Ethereum API communicator module.
- Iteration 3: core functionality for transactions manager module for accessing the three smart contract methods.
- Iteration 4: console interface for accessing the smart contract methods.
- Iteration 5: core functionality for the events manager module for filtering events produced by the smart contract.
- Iteration 6: console interface for retrieving attributes, signatures and revocations.
- Iteration 7: trust store capability for the console.

- Iteration 8: PGP and IPFS capabilities for the transactions and events managers.
- Iteration 9: console capability for adding PGP key attributes over IPFS.

5.0.9 Version Management

Git was used as a version management system for the project's source code. This was used in case any changes in the code had to be rolled back, although there has not been any need for this during the development of the project.

5.1 Smart Contract

The smart contract was implemented in Solidity in the way that is described in the *Design* chapter of the report. The full code of the smart contract is attached to the report in *Appendix A*.

The smart contract implementation is very few lines of code compared to the client implementation, but the smart contract is of equal if not greater importance to the project's implementation as it defines the functionality of the system.

The data entities described in the *Design* chapter of the report were implemented as *structs* in Solidity; a way to define new data types.[12] The data for the entities were stored in *struct arrays* that are manipulated by the methods and easily referenced to by their index number, which also doubles as the numeric ID of each data entity referenced by the users of the system.

For example, the *struct* for the signature attribute and its corresponding *struct array* declaration is:

```
struct Signature {
    address signer;
    uint attributeID;
    uint expiry;
}

Signature[] public signatures;
```

5.2 Client

The client was implemented in Python in the way that is described in the *Design* chapter of the report. The source code is published on GitHub.[1]

The *setuptools* Python package is used to facilitate easy installation of the client using the Python package manager *pip*.

After installation, running the *trustery* command with no argument produces the following help text:

```
Usage: trustery [OPTIONS] COMMAND [ARGS]...

  Ethereum-based identity system.

Options:
  --help Show this message and exit.

Commands:
  add                Add an attribute to your identity.
  ipfsadd            Add an attribute to your identity over IPFS.
  ipfsaddpgp         Add a PGP key attribute to your identity over...
  rawaddattribute    (Advanced) Manually add an attribute to your...
  rawrevokeattribute (Advanced) Manaully revoke your signature of...
  rawsignattribute   (Advanced) Manually sign an attribute about...
  retrieve           Retrieve an attribute.
  revoke             Revoke one of your signatures.
  search             Search for attributes.
  sign               Sign an attribute.
  trust              Trust an Ethereum address.
  trusted            View the list of trusted Ethereum addresses.
  untrust            Untrust an Ethereum address.
```

5.2.1 Raw Commands

The commands *rawaddattribute*, *rawrevokeattribute* and *rawsignattribute* that were not part of the project's original design were added to the interface.

The purpose of these commands is to allow the user to directly pass arguments to the smart contract methods without any data checking or post-processing (such as automatically populating the *datahash* field) by the client.

This was mainly useful during the early development stages of the project for debugging and to test that the smart contract was working as expected.

[1] https://github.com/musalbas/trustery

However, these commands would also be useful to users and other software applications that may directly call the trustery client that wish to implement attributes that are not directly supported by the client. For example, a new type of attribute with a cryptographic proof that the client does not know how to produce, or an attribute that links to off-blockchain data using a protocol that the client does not support.

5.2.2 Managing Attributes

Adding an attribute can be done with the *trustery add* command. The console then prompts the user for the attribute type, identifier and data. For example:

```
$ trustery add
Attribute type: twitter-id
Attribute identifier: foobar
Attribute data []:

Transaction sent.
```

Once the transaction is mined in a block (Ethereum aims towards 12-second block times[21]), the attribute can be searched for using the *trustery search* command:

```
$ trustery search --identifier foobar
Attribute ID #12:
        Type: twitter-id
        Owner: db57ccafaa5198089af9c69499e8d3d67cb463a3 [untrusted]
        Identifier: foobar
        [0 valid signatures]
```

The attribute can then be signed with the *trustery sign* command, with the user specifying the number of days until the signature's expiry:

```
$ trustery sign
Attribute ID: 12
Signature days to expire [365]: 10

Transaction sent.
```

The attribute can then be retrieved with the *trustery retrieve* command (in this case there is no attribute data, as the data is the identifier) and the signature with its validity status is shown:

```
$ trustery retrieve --attributeid 12
```

```
Attribute ID #12:
       Type: twitter-id
       Owner: db57ccafaa5198089af9c69499e8d3d67cb463a3 [untrusted]
       Identifier: foobar
       [1 valid signature]

Signatures for attribute ID #12:
       #3 [valid] by db57ccafaa5198089af9c69499e8d3d67cb463a3 [
           untrusted]

--ATTRIBUTE DATA:
```

The attribute's signature can be revoked with the *trustery revoke* command, with the change being reflected when the attribute is retrieved again:

```
$ trustery revoke --signatureid 3

Transaction sent.

$ trustery retrieve --attributeid 12

Attribute ID #12:
       Type: twitter-id
       Owner: db57ccafaa5198089af9c69499e8d3d67cb463a3 [untrusted]
       Identifier: foobar
       [0 valid signatures]

Signatures for attribute ID #12:
       #3 [revoked] by db57ccafaa5198089af9c69499e8d3d67cb463a3 [
           untrusted]

--ATTRIBUTE DATA:
```

Managing the trust store

The command *trustery trust* can be used to locally mark identities as trusted. This is reflected in all areas of the client when an Ethereum address is displayed. For example:

```
$ trustery trust --address db57ccafaa5198089af9c69499e8d3d67cb463a3

Address db57ccafaa5198089af9c69499e8d3d67cb463a3 trusted.

$ trustery retrieve --attributeid 12
Attribute ID #12:
       Type: twitter-id
```

```
        Owner: db57ccafaa5198089af9c69499e8d3d67cb463a3 [trusted]
        Identifier: foobar
        [0 valid signatures]

Signatures for attribute ID #12:
        #3 [revoked] by db57ccafaa5198089af9c69499e8d3d67cb463a3 [
            trusted]

--ATTRIBUTE DATA:
```

The *trustery untrust* command can also similarly be used to remove identities from the trust store, and the *trustery trusted* command can be used to view the list of identities in the trust store.

5.2.3 PGP

The *trustery ipfsaddpgp* command allows users to publish their PGP keys as an attribute of their identities, with the PGP key data and proof stored on IPFS.

When using the command, the client automatically interacts with the local *gpg* binary to export the public key specified by the user and use its private key to generate the cryptographic proof portion of the attribute.

The cryptographic proof of the attribute is a PGP signature of the Ethereum address that the attribute is to be published from. This proves that the owner of the PGP key has "authorized" a binding between the PGP key to the Ethereum address.

The attribute data is a concatenation of the ASCII armour representative of the PGP key and the PGP signature, such as follows:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1

(data)
-----END PGP PUBLIC KEY BLOCK-----

-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

Ethereum address: (address)
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

(signature data)
```

```
-----END PGP SIGNATURE-----
```

When the *trustery retrieve* command is used on a PGP attribute, the client downloads the attribute data and verifies that the PGP signature is valid and matches the correct Ethereum address and PGP key. The validity of the signature is displayed to the user by the client.

To accomplish this, the client creates a temporary GPG key chain on the local machine, imports the attribute's PGP public key and verifies the signature. The creation of the temporary GPG key chain ensures that any new information processed by GPG is segregated from the user's actual key chain, to prevent any unintended overriding of PGP keys.

The temporary key chain is stored in a directory created using Python's *tempfile* module, which securely creates temporary directories without race conditions and with directory permissions that permit only the user that created the directory to read the directory. After the client is finished processing the attribute's PGP data, the directory and its files are deleted.

Adding and retrieving a PGP key attribute can be achieved as follows:

```
$ trustery ipfsaddpgp --keyid mus@musalbas.com

Transaction sent.

$ trustery retrieve --attributeid 3

Attribute ID #3:
       Type: pgp-key
       Owner: db57ccafaa5198089af9c69499e8d3d67cb463a3 [trusted]
       Identifier: 0x144b59ad818b0e6348709bd061ceb3388f21ea17
       [0 valid signatures]

Proof status for attribute ID #3:
       Valid

Signatures for attribute ID #3:

--ATTRIBUTE DATA:
-----BEGIN PGP PUBLIC KEY BLOCK-----
(...PGP attribute data as shown above...)
```

5.2.4 IPFS

The command *ipfsadd* is identical to the command *add*, except that it stores the attribute data supplied by the user on IPFS. Attribute data is stored in

IPFS blocks, which is a blob of binary data. IPFS provides *get* and *put* API calls for blocks.[22]

The command can be used as follows:

```
$ trustery ipfsadd
Attribute type: twitter-id
Attribute identifier: foobar
Attribute data []:

Transaction sent.
```

5.3 Issues encountered

This section goes into the details of implementation issues encountered in the project, and the steps taken to work around the issues.

5.3.1 Inconsistent Ethereum Python API and Lack of Documentation

One of the biggest issues in implementing the project on the Ethereum smart contract system was that the relatively young age of the Ethereum project meant that there was a massive lack of documentation. This was especially the case for the *pyethereum* library.

Fortunately, the source code for the *pythereum* library is fairly well organized and readable, which meant that it was possible to understand the API methods by reading the code.

However, one recurring issue was an inconsistency in the way data is represented, expected and returned across Ethereum APIs. For example data is returned from the Ethereum JSON RPC in hex format, but *pyethereum* expects it in raw binary format for some parameters and in hex format for other parameters, which lead to a large amount of unintuitive data conversion in the client.

Another example is that the *listen* method in *pyethereum*'s contract translation class is the only method that for an unknown reason prints output to the console by default, which is a cause for confusion while developing a command-line application that prints to the console anyway. As a part of this project's implementation, a minor pull request was submitted and merged to the *pyethereum* project to address this issue.[2]

[2]https://github.com/ethereum/pyethereum/pull/310

5.3.2 Inconsistent Ethereum Python ABI

One of the issues encountered also due to the Ethereum project's young age was a changing Application Binary Interface (ABI). The *pyethereum* package was outdated on the Python Package Index website, leading to early attempts at the implementation of the *pytrustery* client's interaction with Ethereum to not work in the expected way. As a part of this project's implementation, this issue was reported and solved on the *pyethereum* project's issue tracker.[3]

There was also a bug in the Solidity smart contract compiler which meant that indexed event parameters that were longer than 32 bytes were not treated properly, as indexed parameters longer than 32 bytes should be hashed so that they can be represented in 32 bytes.[4] This led to a small change in the project's design to limit attribute identifiers to 32 bytes, and perform the hashing directly in the client if necessary. This solution meant that there is no change to the theoretical functionality of the project.

5.3.3 Gas Limit and PGP

Due to the large size of PGP keys, there was difficulty in directly publishing PGP key data to the blockchain in the test network due to the gas limit specified with transactions; transactions could not be mined.

Regardless, it is not economically sensible to publish such a large amount of data to the blockchain due to the theoretically large gas price.

As a result, the *addpgp* command was renamed to *ipfsaddpgp* as the client only allows PGP attribute data to be stored on IPFS only. This significantly reduces the gas price as only the URI to the PGP data on IPFS needs to be stored on the blockchain.

5.4 Testing

This section describes how the project was tested during its implementation phase.

5.4.1 Private Network

Due to the financial cost associated with paying for the gas price for Ethereum transactions, it is not practical to test the system on the main Ethereum blockchain throughout the main development phase.

[3] https://github.com/ethereum/pyethereum/issues/309
[4] https://github.com/ethereum/solidity/issues/199

Instead, a private Ethereum network and blockchain was setup on a local development machine with the only node and miner being the local machine. The genesis block was programmed to give a large amount of Ether to an Ethereum address with private keys stored on the local machine, to allow for a large amount of test Ether to pay for the gas for test transactions. The smart contract was also published and tested on the private network.

This allowed for the software and transactions to be tested for free on an environment as close as possible to the real Ethereum network.

5.4.2 Unit Testing

Unit tests were developed for the most critical parts of the system.[5]

In particular, unit tests were developed for the code that verifies the proof for PGP attributes. This was achieved by generating attributes that produce different proof statuses (valid, invalid and unknown), and by saving dictionaries that represent these attributes (that the function for verifying proofs expects) in a file in *pickle* format. *Pickle* is a Python object serialization format.[6]

The unit tests can then read the files and deserialize the *pickle* data for usage in tests.

[5]https://github.com/musalbas/trustery/tree/master/pytrustery/test
[6]https://docs.python.org/2/library/pickle.html

Chapter 6

Evaluation

6.1 Project Aims

The aim of this project was to create a system based on a smart contract on the Ethereum blockchain that enables users to manage identities and attributes associated with identities, including cryptographic keys. The system must be decentralized and allows users to sign and verify specific and fine-grained attributes of identities using a web-of-trust model.

The aims of this project have been met for the following reasons:

- Users can manage identities as each Ethereum address represents an identity, and Ethereum addresses can be created by an Ethereum client and trusted or untrusted by the Trustery client.
- Users can manage attributes associated with identities by adding and retrieving attributes.
- Users can manage attributes representing cryptographic keys through the attribute's data field and signify that it has a cryptographic proof through the *has_proof* field.
- The system is decentralized as it uses Ethereum blockchain for the storage of attributes, and optionally allows users to specify any HTTP or IPFS URI for attribute data.
- Users can sign and verify specific attributes through the smart contract's sign method, and the client shows the signatures of attributes by identities when being retrieved to allow users to verify attributes.
- The system uses a web-of-trust model as all users in the system can validly sign and verify other users' attributes.

6.2 Functional Requirements

All of the functional requirements laid out in the specification of the project have been objectively met.

The smart contract was implemented in Solidity and the code objectively satisfies all of the functional requirements.

The client was implemented in Python and the code also objectively satisfies all of the functional requirements.

The requirements were implemented according to the specification.

6.3 Non-functional Requirements

The non-functional requirements that were set out in the design will be evaluated in this section.

6.3.1 Security

One of the non-functional requirements of the project was security. Specifically:

- 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 first point relies on the Ethereum system being secure, which is outside the scope of the project. This project makes the assumption that Ethereum is free from significant security flaws.

Despite this, it is worth analysing the security of Ethereum. One of the main concerns of Ethereum is that if a security issue is discovered in the code that executes EVM code then it may be possible to take control over the Ethereum network or break the rules of a smart contract by broadcasting a transaction that contains malicious EVM code.

This concern can be addressed by the fact that there are at least eight different Ethereum implementations of the Ethereum Virtual Machine in

at least seven different programming languages.[1][2][3][4][5][6][7][8] This means that if a security flaw is found in one of the implementations, as long as the implementation is not used by more than 51% of the network then it should not be possible to "break the rules" of the network.

The second point relies on the security of the client. Attribute integrity has been implemented in the form of the *datahash* parameter in attributes, and cryptographic proofs attached to the attribute's data. Cryptographic proof verification for PGP attributes is also tested with unit tests.

6.3.2 Reliability

One of the project's requirements was that all actions of the smart contract can be performed regardless of the state of the system. The Ethereum system itself provides this reliability as its decentralized nature means that there is no central point of failure. Furthermore, by design smart contracts do not effect the state of other smart contracts unless a smart contract explicitly references another smart contract.

6.3.3 Cross-Platform

One of the project's requirements was that the client should be able to run on most common modern operating systems.

This has been objectively achieved as the client is written in Python, which is a cross-platform non-compiled programming language with implementations for Linux, Windows and Mac.[23]

[1] https://github.com/ethereum/cpp-ethereum
[2] https://github.com/ethereum/go-ethereum
[3] https://github.com/ethereum/ethereumj
[4] https://github.com/ethereum/pyethereum
[5] https://github.com/ethereumjs/ethereumjs-vm
[6] https://github.com/bkirwi/ethereum-haskell
[7] https://github.com/jamshidh/ethereum-client-haskell
[8] https://github.com/Nethereum/Nethereum

6.3.4 Cost Efficiency

6.4 Alternative Cost-Efficient Smart Contract Design

6.5 Limitations

6.5.1 Adaptability

The design of the system is such that all parties referenced by the system must already use the system. For example, if a university wants to issue a degree to an user in the system, the user must first add a degree attribute to their identity before the university can sign it. This increases the adoption barrier for the system.

However, the upside to this is that the user has control over what attributes are or are not attached to their identity.

6.5.2 PGP Cryptographic Proof Binding

Currently, when the user produces a PGP signature to attach to a PGP attribute to prove that a PGP key binds to an Ethereum address, this binding from the PGP attribute to the Ethereum address has no expiration date or revocation mechanism within the smart contract.

This is a problem if for example the private key for the Ethereum address is compromised as users that trust the PGP key bound to the Ethereum address can also trust the identity of the Ethereum address.

In future iterations of this project, this can be addressed by also binding PGP key to the specific attribute and not just the Ethereum address, so that the owner of the identity can simply revoke a self-signature of the attribute.

6.5.3 Privacy

The system is only suitable for the publishing of attributes that the user wishes to make public (such as degree awards). It is not suitable for the publishing of more private identity attributes such as personal address as all attributes can be viewed by anyone in the system and there is no access control.

Future iterations of this project may address this by adding functionality to publish "zero-knowledge" attributes for verification of privately shared data that the user distributes and has control over.

Chapter 7

Professional Issues

Chapter 8

Conclusion

8.0.4 Application of Work in New Areas

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

[15] A. Ronacher. (2016) Click documentation (5.0). [Online]. Available: http://click.pocoo.org/5/

[16] P. Merriam. (2016) ethereum-rpc-client. [Online]. Available: https://pypi.python.org/pypi/ethereum-rpc-client

[17] M. Foord, N. Larosa, R. Dennis, and E. Courtwright. () Reading and writing config files - configobj 5.0.0 documentation. [Online]. Available:

[18] A. Stocker. (2016) ipfs-api : Python package index. [Online]. Available: https://pypi.python.org/pypi/ipfs-api

[19] (2015) The gnu privacy guard. [Online]. Available: https://www.gnupg.org/

[20] V. Sajip. (2015) python-gnupg : Python package index. [Online]. Available: https://pypi.python.org/pypi/python-gnupg

[21] V. Buterin. (2014) Toward a 12-second block time. [Online]. Available: https://blog.ethereum.org/2014/07/11/toward-a-12-second-block-time/

[22] (2016) Ipfs command reference. [Online]. Available: https://ipfs.io/docs/commands/

[23] (2016) Python releases. [Online]. Available: https://www.python.org/downloads/

Appendix A

Smart Contracts

A.1 Main Smart Contract

```
contract Trustery {
    struct Attribute {
        address owner;
        string attributeType;
        bool has_proof;
        bytes32 identifier;
        string data;
        string datahash;
    }

    struct Signature {
        address signer;
        uint attributeID;
        uint expiry;
    }

    struct Revocation {
        uint signatureID;
    }

    Attribute[] public attributes;
    Signature[] public signatures;
    Revocation[] public revocations;

    event AttributeAdded(uint indexed attributeID, address indexed
        owner, string attributeType, bool has_proof, bytes32 indexed
         identifier, string data, string datahash);
```

```
    event AttributeSigned(uint indexed signatureID, address indexed
        signer, uint indexed attributeID, uint expiry);
    event SignatureRevoked(uint indexed revocationID, uint indexed
        signatureID);

    function addAttribute(string attributeType, bool has_proof,
        bytes32 identifier, string data, string datahash) returns (
        uint attributeID) {
        attributeID = attributes.length++;
        Attribute attribute = attributes[attributeID];
        attribute.owner = msg.sender;
        attribute.attributeType = attributeType;
        attribute.has_proof = has_proof;
        attribute.identifier = identifier;
        attribute.data = data;
        attribute.datahash = datahash;
        AttributeAdded(attributeID, msg.sender, attributeType,
            has_proof, identifier, data, datahash);
    }

    function signAttribute(uint attributeID, uint expiry) returns (
        uint signatureID) {
        signatureID = signatures.length++;
        Signature signature = signatures[signatureID];
        signature.signer = msg.sender;
        signature.attributeID = attributeID;
        signature.expiry = expiry;
        AttributeSigned(signatureID, msg.sender, attributeID, expiry
            );
    }

    function revokeSignature(uint signatureID) returns (uint
        revocationID) {
        if (attributes[signatureID].owner == msg.sender) {
            revocationID = revocations.length++;
            Revocation revocation = revocations[revocationID];
            revocation.signatureID = signatureID;
            SignatureRevoked(revocationID, signatureID);
        }
    }
}
```

A.2 Alternative Smart Contract

```
contract Trustery {
    struct Signature {
        address signer;
    }

    uint public attributes;
    Signature[] public signatures;
    uint public revocations;

    event AttributeAdded(uint indexed attributeID, address indexed
        owner, string attributeType, bool has_proof, bytes32 indexed
         identifier, string data, string datahash);
    event AttributeSigned(uint indexed signatureID, address indexed
        signer, uint indexed attributeID, uint expiry);
    event SignatureRevoked(uint indexed revocationID, uint indexed
        signatureID);

    function addAttribute(string attributeType, bool has_proof,
        bytes32 identifier, string data, string datahash) returns (
        uint attributeID) {
        attributeID = attributes++;
        AttributeAdded(attributeID, msg.sender, attributeType,
            has_proof, identifier, data, datahash);
    }

    function signAttribute(uint attributeID, uint expiry) returns (
        uint signatureID) {
        signatureID = signatures.length++;
        Signature signature = signatures[signatureID];
        signature.signer = msg.sender;
        AttributeSigned(signatureID, msg.sender, attributeID, expiry
            );
    }

    function revokeSignature(uint signatureID) returns (uint
        revocationID) {
        if (signatures[signatureID].signer == msg.sender) {
            revocationID = revocations++;
            SignatureRevoked(revocationID, signatureID);
        }
    }
```

```
}
```