SCPKI: A Smart Contract-based PKI and Identity System

Mustafa Al-Bassam
Department of Computer Science
University College London
London, United Kingdom
m.albassam@cs.ucl.ac.uk

ABSTRACT

The Public Key Infrastructure (PKI) in use today on the Internet to secure communications has several drawbacks arising from its centralised and non-transparent design. In the past there has been instances of certificate authorities publishing rogue certificates for targeted attacks, and this has been difficult to immediately detect as certificate authorities are not transparent about the certificates they issue. Furthermore, the centralised selection of trusted certificate authorities by operating system and browser vendors means that it is not practical to untrust certificate authorities that have issued rogue certificates, as this would disrupt the TLS process for many other hosts.

SCPKI is an alternative PKI system based on a decentralised and transparent design using a web-of-trust model and a smart contract on the Ethereum blockchain, to make it easily possible for rogue certificates to be detected when they are published. The web-of-trust model is designed such that an entity or authority in the system can verify (or vouch for) fine-grained attributes of another entity's identity (such as company name or domain name), as an alternative to the centralised certificate authority identity verification model.

1. INTRODUCTION

The secure operation of SSL/TLS relies on a set of trusted Certificate Authorities (CAs) to authenticate public keys [5]. In practice, the set of trusted CAs are bundled into operating systems and web browsers. Consequently, the Public Key Infrastructure (PKI) is centralised as only CAs chosen by operating system and web browser vendors may issue universally 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[1].

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].

Pretty Good Privacy (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 [6].

The web-of-trust model is a first step towards a decentralised 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 centralised keyservers[2] that are used to query for public keys.

Ideally, a PKI for PGP would be be fully decentralised and not rely on centralised servers. Centralised keyservers act as a central point of failure that also allow for exclusion or replacement of keys by a third party.

Most importantly, an immutable blockchain-based decentralised 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.

The X.509 standard for certificates on the Internet provides scope for a wide range of identity attributes to be embedded in certificates [7]. Identity attributes include information such as phone number, address and name. This provides a way for certificate authorities to vouch for the identity attributes 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 forged, unlike cryptographic signatures.

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 web-of-trust to trust the branch of the government that is responsible for giving universities degree-awarding

[1] https://wiki.mozilla.org/CA:How_to_apply

[2] Such as pgp.mit.edu.

BCC'17 April 02-02 2017, Abu Dhabi, United Arab Emirates

ACM ISBN 978-1-4503-4974-1/17/04.

DOI: http://dx.doi.org/10.1145/3055518.3055530

powers, and work through the remaining chain-of-trust.

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

This is 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.

2. BACKGROUND

2.1 Blockchain

The concept of the blockchain was first introduced in the Bitcoin electronic cash system [8]. 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 is below a cetain value. This requires processing power, and so as long as the majority of the processing 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 most proof-of-work is the correct one, and so an authority that does not have the majority of the network's processing power is unlikely to outpace the creation of blocks of the rest of the network.

The open 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 useful tool to build decentralised and transparent applications where records cannot be hidden or corrupted by a third party.

This is particulaly useful in the context of building a transparent PKI, as rogue certificates or identities would be universally visible.

2.2 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 [4].

Since the creation 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 [10]. 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" [1].

One of the potential applications of smart contracts discussed in the white paper is identity and reputation systems. For example, a smart contract can be created for mapping domain names to IP addresses to provide a decentralised domain name registration system. Namecoin is an example of such a system that uses a Bitcoin-like blockchain[3].

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 [10].

Each smart contract has functions that have "gas" costs associated with it depending on how many computational steps or storage space they require which are paid for using Ether—Ethereum's internal currency, and smart contracts can call functions in other smart contracts [10].

Ethereum smart contracts can emit "events", an abstraction of the Ethereum logging and event-watching protocol [2]. Events can have up to three indexes, and can be watched and filtered efficiently by Ethereum clients.

The primary proposition of SCPKI 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.

2.3 Web of Trust

An alternative to the centralised chain of trust certificate authority model of PKI is the web of trust.

The concept of the web of trust was first described in 1992 in the manual for PGP 2.0: "As time goes on, you will accumulate keys from other people that you may want to designate as trusted introducers. Everyone else will each choose their own trusted introducers. And everyone will gradually accumulate and distribute with their key a collection of certifying signatures from other people, with the expectation that anyone receiving it will trust at least one or two of the signatures. This will cause the emergence of a decentralised fault-tolerant web of confidence for all public keys" [9].

In a web of trust, there are no certificate authorities. Instead any user of the system can sign each other's public key, which means that there is no certificate authority in the system that is "too big to fail" as public keys are by design intended to have multiple signatures. This means that if one signer is compromised and the signer's key is revoked, the impact on the trust network is limited.

2.4 IPFS

Smart contract systems like Ethereum are useful as a high-integrity progammable database layer for decentralised applications. However due to gas costs, and it is not economically practical to store large amounts of data on the Ethereum blockchain.

InterPlanetary File System (IPFS)[4] has become a popular storage layer for decentralised applications. It is a peer-to-peer data distribution protocol where nodes in the IPFS network form a distributed file system.

[3] http://namecoin.org/

[4] https://ipfs.io/

Data in IPFS is addressed by its cryptographic hash and so links always stays the same regardless of which nodes serve the data. This makes it ideal for blockchain applications, as it makes it possible to address large amounts of data from transactions in the blockchain using permanant and immutable IPFS links.

3. DESIGN

SCPKI is 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.

The design of SCPKI contains two primary components: the smart contract—which dictates the protocol of the system and acts as an interface to the blockchain for the management of identities and attributes, and the client—which interacts with the smart contract and other systems such as IPFS to allow users to fully utilise the system by allowing them to search for and filter attributes.

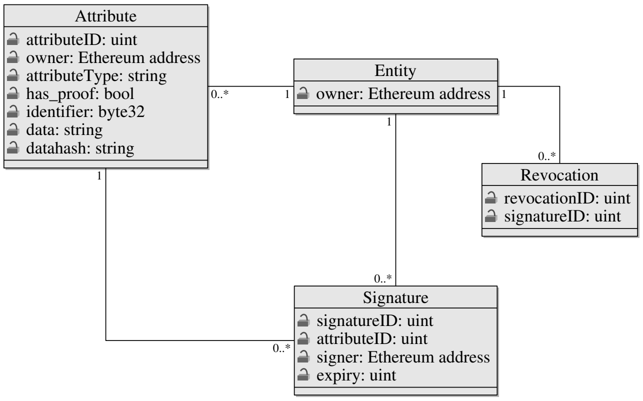

Figure 1: Entity-relationship diagram for the SCPKI smart contract.

The smart contract of SCPKI centers around the entity, which publishes a set of attributes, signatures, and revocations on the blockchain for its identity. Each entity is represented by an Ethereum address—which is controlled by a private key or smart contract that the entity has control over.

Although publishing an attribute to an entity's identity binds the identity to the attribute, attributes that represent cryptographic keys can also be reverse-binded to the identity, creating a double-binding. This can be done by publishing a "binding proof" that consists of a cryptographic signature of the entity's Ethereum address using the cryptographic key represented by the attribute, proving that the owner of a private key is associated with an Ethereum address. This is useful for transfering trust relationships between different cryptographic keys: for example if a user A already trusts a user B's PGP key, and user B adds their PGP key to their SCPKI identity with a binding proof, then user A can also trust user B's SCPKI identity without asking for more information.

Due to the expensive gas costs associated with Ethereum storage, SCPKI is designed to allow users to store large attribute data (such as PGP keys) off the blockchain (such as on IPFS) to save costs, while maintaining the authenticity of the data by providing a cryptographic hash of the data with the attribute on the blockhain.

Two vesions of the smart contract can exist, a full version that is designed in such a way where its information (attributes, signatures and revocations) can be programatically accessed by other smart contracts, and a lighter version where it cannot. The trade-off is that the lighter version has lower gas costs because there is no need to store attribute data within the contract itself for retrieval by other contracts, but simply emit events on the blockchain for clients to watch, but is less extensible by other smart contracts that interact with it.

3.1 Adding Attributes

The *AddAttribute* function allows entities to add attributes to their identity. As input, it takes the following parameters.

Attribute type. The type of attribute that is represented, for example "PGP key" or "name".

Identifier. A unique identifier for the attribute if it has one, such as a PGP fingerprint.

Data. The raw data of the attribute, or an URI where the data can be downloaded off the blockchain. The data also includes the binding proof of the attribute, if applicable.

Data hash. If the data is stored off the blockchain, the cryptographic hash of the data for authenticating the data, if necessary. This is not necessary if the data is stored on IPFS, as IPFS links contain the hash of the data.

Has proof. True if it is a cryptographic attribute that has a binding proof, otherwise False.

The function also generates a unique automatically incrementing *attribute ID* for the new attribute, which is used to reference the attribute. The *owner* of the new attribute is the Ethereum address of the transaction that called the smart contract function.

An *AttributeAdded* event is then logged with all of the input parameters of the function call as well the attribute ID and attribute owner, for SCPKI clients to pick up. The event has three indexes that can be filtered by in the client: the attribute ID, the attribute owner and the attribute identifier.

In the full version of the smart contract, the function stores the data of all attributes in an array of attributes, which can be accessed by other smart contracts by ID. The attribute ID is the index of the attribute in the array. In the light verson of the smart contract, the function only stores and increments the latest attribute ID.

3.2 Signing Attributes

The *SignAttribute* function allows entities to sign ("vouch for") the attributes of any entities. It takes the following parameters.

Attribute ID. The ID of the attribute being signed.

Expiry. A UNIX timestamp representing the time at which the signature should no longer be considered valid.

The function also generates a unique automatically incrementing *signature ID* for the new signature. The *signer* of the new signature is the Ethereum address of the transaction that called the function.

A *SignatureAdded* event is then logged with all of the input parameters of the function call as well the signature ID and signer. The event has three indexes that can be filtered by in the client: the attribute ID, the signature ID and the signer.

In the full version of the smart contract, the function stores the data of all signatures in an array of signatures, which can be accessed by other smart contracts by ID. The signature ID is the index of the signature in the array. In the light verson of the smart contract, the function stores only stores the data of the signers of the signatures in the signatures array. This information is necessary in order for the contract's signature revocation function to only allow the signers of signatures to revoke them.

3.3 Revoking Signatures

The *RevokeSignature* function allows entities to revoke their own signatures It takes one parameter, **Signature ID**—a reference to the signature to be revoked. The function also generates a unique automatically incrementing *revocation ID* for the new revocations.

A *SignatureRevoked* event is then logged with the signature ID the revocation ID, which are both indexes.

In the full version of the smart contract, the function stores the data of all revocations in an array of revocations, which can be accessed by other smart contracts by ID. The revocation ID is the index of the revocation in the array. In the light verson of the smart contract, the function only stores and increments the latest revocation ID.

The *RevokeSignature* function only revokes signatures if the Ethereum address calling it is also the signer of the signature to be revoked; the signature is checked to ensure that the signer matches the sender of the transaction calling the function. The procedure for this is shown in *Algorithm 1.*

Algorithm 1 Revocation Procedure

```
procedure REVOKESIGNATURE
    if $transaction.sender = signatures[signatureID].signer$ then
        $revocationID \leftarrow revocations.length + 1$
        $revocation \leftarrow revocations[revocationID]$
        $revocation.signatureID \leftarrow signatureID$
        SIGNATUREREVOKED(revocationID, signatureID)
        return $successful$
    else
        return $failed$
    end if
end procedure
```

4. IMPLEMENTATION

A working prototype of SCPKI is implemented, and available as an open-source project[5]. The smart contracts are implemented in 90 lines of Solidity, a JavaScript-like language for writing Ethereum smart contracts that are compiled to EVM code. The full and light contracts are available in *Appendix A* and *Appendix B*, respectively.

A corrosponding command-line Python client was implemented in 844 lines of code. The client allows users to manage their identity by adding attributes, signing attributes and revoking signatures. It connects to the local Ethereum client's JSON RPC to send and recieve data to and from the blockchain.

The client allows users to retrieve and validate attributes and their signatures/revocations from the blockchain, and search for attributes filtered by ID, type or identifier.

The client also interfaces with IPFS and PGP, to allow clients to seamlessly add and retrieve attributes that have data stored off the blockchain on IPFS, and to allow clients to publish their PGP key attribute with an automatically generated binding proof to their identity. The client can also automatically verify binding proofs of the PGP key attributes of other users.

4.1 Costs

Using the implementation, the gas costs of using SCPKI can be analysed. As of January 2017, 1 gas = 0.000000018 ether[6], and 1 ether = \$10.57[7].

Table 1 and *Table 2* show the gas costs for adding attributes of various sizes with and without IPFS, on the full and light smart contract. When not using IPFS, the gas cost of a transaction to add an attribute grows linearly (by around \$0.015 per 100 bytes for the full contract and by around \$0.001 per 100 bytes for the light contract) with the size of the data of the attribute. When using IPFS, it remains static because the size of the link to data on IPFS stays the same regardless of the size of data.

The light smart contract shows a significant cost improvement—just \$0.007 to add an attribute over IPFS compared to \$0.032 without; a 72% reduction.

Table 3 and *Table 4* show the gas cost for publishing the smart contract on the blockchain, signing attributes and revoking signatures. These operations have static gas costs as they do not have data of variable length as input. By far the biggest cost (over a tenth of a dollar for the full contract) is the cost of publishing the contract on the blockchain. This is a one-time cost necessary for users to start using the contract. All other operations are relatively cheap at \$0.012 or under per operation, for both types of contracts.

5. CONCLUSIONS AND FUTURE WORK

This paper set out to realise a decentralized public key infrastructure system that utilises the transparency of the blockchain and has fine-grained attribute management for the web of trust.

This has been achieved in the form of a smart contract and a command-line client for using the smart contract, with a working prototype implemented.

As the Internet scales, rogue certificate attacks become more common, and organizations demand more forgery-proof identity verification techniques, the need for transparent public key infrastructure systems will grow.

Some limitations and potential future work of SCPKI are discussed below.

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.

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

[6] https://ethstats.net/

[7] https://coinmarketcap.com/

Data (bytes)	Gas	Gas in USD	Gas (IPFS)	Gas in USD (IPFS)
100	213782	$0.041	170215	$0.032
200	281595	$0.056	170215	$0.032
300	349473	$0.066	170215	$0.032
400	417287	$0.079	170215	$0.032
500	485037	$0.092	170215	$0.032

Table 1: Gas used for adding attributes in the full smart contract.

Data (bytes)	Gas	Gas in USD	Gas (IPFS)	Gas in USD (IPFS)
100	41789	$0.008	38372	$0.007
200	49377	$0.009	38372	$0.007
300	57030	$0.011	38372	$0.007
400	64619	$0.012	38372	$0.007
500	72144	$0.014	38372	$0.007

Table 2: Gas used for adding attributes in the light smart contract.

Operation	Gas	Gas in USD
Publishing contract	729752	$0.139
Signing attribute	62799	$0.012
Revoking signature	22120	$0.004

Table 3: Gas used for other operations in the full smart contract.

Operation	Gas	Gas in USD
Publishing contract	353385	$0.067
Signing attribute	49904	$0.009
Revoking signature	22120	$0.004

Table 4: Gas used for other operations in the light smart contract.

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 the system 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.

6. REFERENCES

[1] A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper/784a271b596e7fe4e047a2a585b733d631fcf1d4, 2016.

[2] Ethereum contract abi. https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI/e6077256597058bd257f75740955caa10624086d, 2017.

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

[4] A. M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Crypto-currencies*. O'Reilly Media Incorporated, 2014.

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

[6] S. Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly & Associates, 1994.

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

[8] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[9] H. F. Tipton. *Official (ISC)2 Guide to the SSCP CBK, Second Edition*. Auerbach Publications, Boston, MA, USA, 2nd edition, 2010.

[10] G. Wood. Ethereum: A secure decentralised generalised transaction ledger (eip-150 revision). https://github.com/ethereum/yellowpaper/raw/2c6fba1400e321734ccec19cb5d9cb32a51ffc44/paper.pdf, 2017.

APPENDIX

A. FULL SMART CONTRACT

```
contract SCPKI {
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);
    }
}
}
```

B. LIGHT SMART CONTRACT

```
contract SCPKI {
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);
    }
}
}
```