SmartDNSPKI: A blockchain based DNS and PKI

Shashee Kumari

SmartDNSPKI: A blockchain based DNS and PKI

DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology
in
Computer Science

by

Shashee Kumari
[Roll No: CS-1525]

under the guidance of

Dr. Sushmita Ruj
Assistant Professor
Cryptology and Security Research Unit

Indian Statistical Institute
Kolkata-700108, India

July 2017

To my family and friends

CERTIFICATE

This is to certify that the dissertation entitled **"SmartDNSPKI: A blockchain based DNS and PKI"** submitted by **Shashee Kumari** to Indian Statistical Institute, Kolkata, in partial fulfillment for the award of the degree of **Master of Technology in Computer Science** is a bonafide record of work carried out by her under my supervision and guidance. The dissertation has fulfilled all the requirements as per the regulations of this institute and, in my opinion, has reached the standard needed for submission.

Sushmita Ruj
Assistant Professor,
Cryptology and Security Research Unit,
Indian Statistical Institute,
Kolkata-700108, INDIA.

Acknowledgements

I would first like to thank my advisor, **Sushmita Ruj**, Cryptology and Security Research Unit, Indian Statistical Institute, Kolkata, for guiding, supporting, and motivating me over the year. You have set an example of excellence as a researcher and a mentor to your students.

I would also like to thank Prabal Banerjee, Junior Research Fellow, Indian Statistical Institute, Kolkata, and Ram Govind Singh, Junior Research Fellow, Indian Statistical Institute, Kolkata, for helping me with the Ethereum platform.

My deepest thanks to all the teachers of Indian Statistical Institute, for their valuable suggestions and discussions which added an important dimension to my research work.

Finally, I am thankful to my family, friends, and all those who helped me complete this project, directly or indirectly.

Shashee Kumari
Indian Statistical Institute
Kolkata - 700108 , India.

Abstract

PKI is an infrastructure used to create, store, manage, revoke, and distribute Digital certificates which bind a public key to an entity. PKI is implemented broadly using two approaches: first approach is a centralised system with Certificate Authorities (CAs) playing the crucial role of certifying an entity and publishing the certificate of the certified entity. This introduces a central point of failure in the system in the form of trusted CAs. Moreover, Certificate Authorities are not publicly auditable making it more difficult to detect a fraud CA. The second approach is “Web-of-Trust (WoT) ”, which is a decentralised system of certifying the entities and has no trusted third party. The participating entities can certify themselves and get attestations from other participants who vouch for their certificates and are trusted by other users. However, a participant needs a trusted introducer to enter the system. To overcome these problems, few blockchain-based PKIs have been proposed to make the process of certificate issuance, updation and revocation publicly auditable and unalterable.

Google suggested use of append-only ledgers to make the activity of certificate issuance publicly auditable. Use of a publicly auditable, append-only ledger, which is otherwise very useful, comes with a number of privacy-related challenges. One of those challenges is registering a certificate for a private subdomain. Certificate transparency has extended support to register redacted domain names in CT logs. However, domain name redaction has many weaknesses. Use of wildcard certificates for securing subdomains is also ambiguous and insecure. Other blockchain-based PKIs also do not have any method to support private subdomains in a secure way. We have proposed SmartDNSPKI: a blockchain-based DNS and PKI, which provides all the functionalities of a DNS and a PKI preserving the privacy of private subdomains. Our scheme does not reveal any information about the private subdomains, apart from the fact that a private subdomain has been registered for the domain. To support the linkability between the certificate owner and the domain owner, the two smart contracts interact with each other. We have implemented our solution on Ethereum platform.

Contents

List of Tables

Chapter 1

Introduction

1.1 PKI

A **public key infrastructure (PKI)** [6] is a comprehensive system for the creation, storage, and distribution of digital certificates which are used to verify that a particular public key belongs to a certain entity. The certified entity could be a person, a group, an agency, an organization, a business, etc. The PKI creates digital certificates which map public keys to entities, securely stores these certificates in a central repository and revokes them if needed. It establishes and maintains a trustworthy networking environment by providing key and certificate management services that enable encryption and digital signature capabilities across applications. A PKI consists of:

- A **Certificate Authority (CA)**: It is a trusted third party which stores, issues and signs the digital certificates.

- A **Registration Authority (RA):** It verifies the identity of entities requesting their digital certificates to be stored at the CA

- A **central directory:** It is a secure location in which CAs store and index keys

- A **certificate management system:** It manages things like the access to stored certificates or the delivery of the certificates to be issued.

- A **certificate policy:** It specifies all the actors in the PKI, their roles and duties.

1.1.1 Types of certifications

Broadly speaking, PKI has two main implementations: Certificate Authorities (CAs), Web of Trust (WoT).

1.1.1.1 CERTIFICATE AUTHORITY

A Certificate Authority(CA) [1] is an entity which acts as a trusted third party for issuing Digital Certificates which establish ownership of a public key by the named Subject of the Digital Certificate. The CA validates the identity of a Certificate holder and signs the Certificate to confirm that it hasn't been forged or altered in any way. The certificate contains the name of the certificate holder, a serial number, expiration dates, a copy of the certificate holder's public key (used for encrypting messages and digital signatures) and other relevant information. To provide evidence that a certificate is genuine and valid, it is digitally signed by a root certificate belonging to a trusted certificate authority. Operating systems and browsers maintain lists of trusted CA root certificates so they can easily verify certificates that the CAs have issued and signed.

Incorporation of CAs in PKI introduces a central point of failure in the system. If the CA can be subverted, then the security of the entire system is lost, potentially subverting all the entities that trust the compromised CA. There have been numerous incidents when a CA, knowingly or unknowingly, has issued a fake certificate for an entity which was used by the hackers for carrying out Man-in-the-middle attacks against the entity. In 2001, certificate authority VeriSign issued two certificates to a person claiming to represent Microsoft. The certificates had the name "Microsoft Corporation", so they could be used to spoof someone into believing that updates to Microsoft software came from Microsoft when they actually did not. Microsoft and VeriSign took steps to limit the impact of the problem [8]. In 2011, fraudulent certificates were obtained from Comodo [25] and DigiNotar [14, 3] allegedly by Iranian hackers. In 2012, it became known that Trustwave issued a subordinate root certificate that was used for transparent traffic management (man-in-the-middle) which effectively permitted an enterprise to sniff SSL internal network traffic using the subordinate certificate [7].

Google's Certificate Transparency [20] is an elegant way of keeping a check on this kind of misbehavior by CAs and provides a strong defense against misissuance. However, there are certain privacy issues which remain unaddressed by the current design of Certificate Transparency. One of them is incompatibility with private subdomains. There are many enterprises which do not want to reveal the name of their private subdomains but they want to use a public CA to issue certificates for those internal subdomains. Currently two solutions for this privacy requirement are available, namely Wildcard Certificates [23] and Domain Label Redaction [2] both of which are not encouraged by Certificate Transparency because of the shortcomings with these two techniques as discussed in [23, 2]. Both these techniques have security flaws and are ambiguous.

Keeping in view the weaknesses in the current centralized PKI, many researchers have suggested the use of blockchains to decentralize the same. However, none of these address the privacy concern of organizations with private subdomains.

1.1.1.2 WEB OF TRUST

An alternative approach to the problem of public authentication of public key information is the Web-of-Trust scheme [16, 9], which uses self-signed certificates and third party attestations of those certificates. This concept was first put forth by PGP creator Phil Zimmermann in 1992 in the manual for PGP version 2.0. It is a decentralised scheme in which an user signs her certificate and gets it attested by other users, who by that act, endorse the association of that public key with the person or entity listed in the certificate. Every user chooses her own trusted introducer and gradually accumulates and distributes with her 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 system is decentralised, and hence, free from a central point of failure. However, it introduces a barrier for new or remote users as it is difficult for them to find enough endorsers and meet them in-person for getting their new certificates attested by them. Also, there is no key recovery scheme in this model and loss of private key is dealt with the use of “designated revokers ”who have the task of revoking the lost or the compromised key on the behalf of the certificate owner.

1.2 DNS

Domain Name System (DNS) [24] is a distributed and hierarchical naming system for resources such as computers, servers and other devices on Internet or private network which helps translating domain names to IP addresses so that the devices can communicate with each other. Currently this is implemented in the form of a huge distributed database across the world. After a domain is successfully registered, it needs to acquire a digital certificate certifying its public key which can be used for secure communication over the Internet. Currently, DNS and PKI function independent of each other and separate entities are responsible for setting the protocols for each system. There are few blockchain-based DNS also, most prominent of them being Namecoin.

1.3 Blockchain

A blockchain is a distributed database that is used to maintain a continuously growing list of records, called blocks. Each block contains a timestamp and a link to a previous block. A blockchain is typically managed by a peer-to-peer network collectively adhering to a protocol for validating new blocks. By design, blockchains are inherently resistant to modification of the data. Once recorded, the data in any given block cannot be altered retroactively without the alteration of all subsequent blocks and the collusion of the network. Functionally, a blockchain can serve as an open, distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way. The ledger itself can also be programmed to trigger transactions automatically. Blockchain became popular with the advent of Bitcoin in 2008. However, Bitcoin does not support a Turing complete coding language.

1.4 Our contribution

We propose *SmartDNSPKI*: a blockchain-based DNS and PKI which can be implemented on any blockchain supporting smart contracts. It supports the registration and update of domains as well as their public and private subdomains. It also allows a domain owner to register, update, and revoke the certificates for her domain and subdomains which are registered on the blockchain. To deal with the privacy issue of private subdomains being publicly visible on the blockchain, we suggest a method to hide the private subdomain name using a commitment scheme which ensures only the users visiting the subdomain know the name of the private subdomain. Other users viewing the blockchain can know nothing about the subdomain except for the fact that a private subdomain is registered for a domain. Public subdomains are registered without the modification of the subdomain name. A digital certificate for a domain and all its subdomains is registered only after validating the domain owner and the registered subdomain. Use of registered cryptographic keys to sign all the requests for registration, update, and revocation of data stored on the blockchain ensures the credibility of the entity sending such requests.

1.5 Thesis Outline

Rest of the sections of this thesis are structured as follows: Related work is briefly discussed in Chapter 2. Content of Chapter 3 is divided into five parts: the cryptographic construction used in our scheme, participants in the whole setup, our design in detail, certificate validation during client server interaction, and security of the proposed scheme. Chapter 4 contains a brief introduction to Ethereum, the blockchain-based platform on which we have implemented our scheme followed by the implementation and performance details of the smart contracts that have been implemented and tested on Ethereum testnet. Thesis ends with Chapter 5 in which we conclude our work and discuss the future directions to our work.

Chapter 2

Related work

In this chapter, we have discussed the existing blockchain-based and other DNS and PKI which are closely related to our work.

2.1 Namecoin

Namecoin [19, 5] is a cryptocurrency designed to act as a decentralized DNS for .bit addresses. It is the first fork of Bitcoin, the distributed digital cryptocurrency, with minor changes and extra functionality and uses the same proof of work algorithm as Bitcoin. The most important change is that Namecoin is able to store data other than just transactions in the blockchain. Each Namecoin record consists of a key and a value which can be up to 520 bytes in size. Each key is actually a path, with the namespace preceding the name of the record.

2.2 Certcoin

Certcoin [18] is a public and decentralized authentication scheme which implements the idea of maintaining a public ledger of domains and their associated public keys to ensure identity retention. It is a branch of Namecoin and is built on top of it using it as a public “bulletin board”where all the transactions are mined together with the transactions of Namecoin taking advantage of merged mining. Certcoin supports all the functionalities of a PKI like registration, update and revocation of a certificate.

2.3 Blockstack

Blockstack [11] is the first implementation of a decentralized DNS system on top of the Bitcoin blockchain combining DNS functionality with public key infrastructure and is primarily meant to be used by new blockchain applications. It has a portable architecture meaning that It is designed to be able to read and write data to any blockchain and the logic for operating the domain name system is decoupled from the logic of the underlying blockchain. Blockchain implements Control plane and Data plane separately, i.e., it decouples security of name registration and name ownership from data availability of values associated with names. The Control plane is responsible for registering human-readable names and creating (name, hash) bindings. It also defines the protocol for establishing ownership of names, which are owned by cryptographic key-pairs. The control plane consists of a cryptocurrency blockchain and a logically separate layer on top, called a "Virtual blockchain". The data plane is responsible for data storage and availability. It consists of (a) routes for discovering data, and (b) external storage systems for storing data (such as Amazon S3, IPFS, or Syndicate).

2.4 CONIKS and EthIKS

CONIKS [22] is an end-user key verification service which enables a centralized service provider to maintain an auditable yet privacy-preserving directory of users'public keys. It obviates the need for global third-party monitors and enables users to efficiently monitor their own key bindings for consistency. CONIKS users and providers can collectively audit providers for non-equivocation using gossip protocol. EthIKS [13] is the implementation of CONIKS on Ethereum platform with minor modifications, leveraging the similar data structure used in Ethereum as well as the consensus protocol of Ethereum that could obviate the need of separate gossip protocol to prevent equivocation and ensure consistency.

2.5 Certificate Transparency

Certificate Transparency [20] is a system proposed by Google, comprising of Certificate Logs, Monitors and Auditors for making the issuance and existence of Digital certificates transparent. Certificate Logs are cryptographically assured, publicly auditable, append-only database of certificates which append new certificates to an ever-growing Merkle hash tree. Monitors are publicly run servers that periodically contact all of the log servers and watch for suspicious certificates. For example, monitors can tell if an illegitimate or unauthorized certificate has been issued for a domain, and they can watch for certificates that have unusual certificate extensions or strange permissions, such as certificates that have CA capabilities. Auditors are lightweight software components that verify If a log is behaving properly and whether a particular certificate appears in a log.

2.6 Certificate Transparency with Privacy

Certificate Transparency poses some privacy challenges to the users. Certificate Transparency auditing can compromise users' browsing privacy. Also Certificate logs do not support logging of certificates for private subdomains without revealing the private subdomain's name. Certificate Transparency with Privacy [17] is an extension to the existing approach of Certificate Transparency with support for auditing the logs without compromising users'browsing privacy. It also supports logging of certificates for private subdomains without revealing the name of the subdomain.

2.7 IKP

IKP [21] is a blockchain-based PKI enhancement that offers automatic responses to CA misbehavior and incentives for those who help detect misbehavior. IKP's decentralized nature and smart contract system allows open participation, offers incentives for vigilance over CAs, and enables financial recourse against misbehavior. In this system, domain owners and the CAs adhere to some Domain registration policies and Reaction policies and the misbehaving CAs are punished as per the terms of the reaction policies.

2.8 SCPKI

Smart Contract based PKI and Identity System [10] 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 centralized certificate authority identity verification model.

Chapter 3

Design

We first present the cryptographic construction used in our design and then discuss the design of our scheme in detail.

3.1 Commitment Scheme

In this scheme, we need a commitment scheme with good hiding property to preserve the privacy of the subdomains. Intuitively a commitment scheme is a two-party protocol between a sender P and a receiver V in which after the sender P commits to a value b at hand, (1) the sender P cannot change the committed value (this is known as the binding property); and (2) the receiver V learns nothing about the value b unless the sender P reveals the value himself (this is known as the hiding property). A commitment scheme consists of three phases:

(a) CK $\leftarrow$ Setup(1^{k}) generates the public commitment key.

(b) *(c,d)*$\leftarrow$ $\mathrm{Commit}_{\mathrm{CK}}(m)$ generates a commitment(c)/decommitment(d) pair for any message $m \in M$.

(c) $\mathrm{Open}_{\mathrm{CK}}(c, d)$ $\rightarrow$ m' $\cup$ $\{\perp\}$ where $\perp$ is returned if c is not a valid commitment to any message.

3.1.1 Pedersen commitment scheme

Pederesen commitment scheme is based on a specific number theoretic assumption - the discrete log assumption.

a) Setup : Sender chooses the following values:

(1) large primes p and q such that q divides *p-1*.

(2) g, a generator of the order-q subgroup of Z_p^*.

(3) a, a random secret from Z_q, and $h = g^a \ mod\ p$.

The values p, q, g and h are public, while a is secret.

b) Commit : For any message $m \in Z_p$, sender chooses a random $r \in Z_q$, computes commitment $c = g^m h^r$ and sends it to the receiver.

c) Decommit : Sender reveals m and r to the reciever. Receiver verifies that $c = g^m h^r$.

3.2 Participants

In this section, we describe the active participants in this scheme which are as explained below:

3.2.1 Blockchain

It is the append-only ledger on which all the transactions and data regarding domains and their subdomains, certificates of the domains and their subdomains are stored. The validity of the blockchain is maintained by a consensus algorithm followed by the miners who validate and execute the transactions and append blocks of transactions to the blockchain. We have chosen Ethereum blockchain for our purpose.

3.2.2 Domain Owner

A domain name [4] represents an Internet Protocol (IP) resource, such as a personal computer used to access the Internet, a server computer hosting a web site, or the web site itself or any other service communicated via the Internet. A domain owner is a person, company or entity who owns or holds a domain name. In our scheme, such an entity is represented by an address, derived from a key pair *(pk, sk)*, which registers a domain name on the blockchain.

3.2.3 Certificate Owner

A certificate owner is an entity which registers a certificate for a domain and its subdomains. In our scheme, this entity is same as the domain owner, as a certificate owner is validated against a domain owner before the certificate is registered.

3.2.4 Users

Entities which use the service of Domain Name System and the Public Key Infrastructure on Internet. These users request the DNS to resolve the domain name and then request PKI for a digital certificate of the domain for validating the entity with which she is interacting. In our scheme, these users fetch these information from the data stored on the blockchain.

3.3 Design

SmartDNSPKI is a decentralised, smart-contract based scheme for DNS and PKI. It facilitates domain/subdomain registration and update as well as registration, update, revocation of digital certificates for each domain and subdomain in a privacy preserving fashion, providing cryptographic linking between domain owner and the domain certificate owner.

3.3.1 DNS

Each domain owner registers all the details of her domain such as its name, IP address or nameserver, and validity information along with an update address which is used to send transactions for registering subdomains and updating domain and subdomains (both public as well as private) later. This address is also used for sending transactions for updating, and revoking digital certificates for the domain as well as its subdomain. Registration of a domain is divided into two parts. In first part, the domain owner sends the hash of the domain to protect it from getting stolen by other active users on the network before registration. In second part, user sends the actual domain name whose hash is matched with the hash sent in the first step. Upon success, the address which has sent these transactions becomes the owner of the domain. Owner of the domain can be changed by transferring the domain to another address.

3.3.2 Structure of a DNS record in SmartDNS contract

A DNS record for a domain is stored in a structure *Domain* which stores the IP address or the nameserver *addr*, validity information *valid_from* and *valid_to*, owner address *owner_addr*, update address *update_addr* and two arrays *pubsd* and *privsd* for storing the information about registered public and private subdomains respectively. Details of a subdomain are stored in another structure *Subdomain* which stores the subdomain name(commitment, in case of private subdomains) and its IP address or nameserver *addr*. Structure of a DNS record is shown in table 3.1.

3.3.3 PKI

Domain owner registers certificates for all its domains and subdomains to the blockchain. A certificate for a domain stores information such as the public key of the domain, its validity information, and its revocation status. It also stores certificates for both public as well as private subdomains of the domain. Certificate of a subdomain stores registration status and revocation status of the subdomain. The certificate of a domain and its subdomains can be revoked separately by sending revocation transaction from the update address of the domain.

3.3.4 Structure of a certificate in SmartPKI contract

A certificate for a domain is stored in a structure *Certificate* which stores the public key pk_{domain_name} of the domain, its validity information, *valid_from* and

Domain Record	
Domain Name	
Field	**Use**
addr	IP address or nameserver address of the domain
valid_from	Specify the start period of the validity of the Domain
valid_to	Specify the end period of the validity of the Domain
owner_address	Specify the address of the domain owner on the blockchain
update_address	Specify the address which can be used to update the domain record and certificates
count_pubsd	Specify the number of public subdomains of the domain registered on the blockchain
count_privsd	Specify the number of private subdomains of the domain registered on the blockchain

Public Subdomains		**Private Subdomains**	
SD Name(1)		**SD Commitment(1)**	
Field	**Use**	**Field**	**Use**
name	Store the name of the public subdomain	comm	Store the commitment to the private subdomain name
addr	Store the address of the subdomain	addr	Store the nameserver address
. . .		. . .	
SD Name(n)		**SD Commitment(n)**	
Field	**Use**	**Field**	**Use**
name	Store the name of the public subdomain	comm	Store the commitment to the private subdomain name
addr	Store the address of the subdomain	addr	Store the nameserver address

Table 3.1: Structure of a Domain Record in the contract storage

valid_to, its revocation status *revoked*, and two mappings *public_sd* and *private_sd* for storing the certificates of the public and private subdomains respectively. Certificates of the subdomains again is a structure *Certificate_SD* the registered status *registered* and revocation status *revoked*. Validity of the subdomain's certificate is same as that of the domain. Structure of a Domain certificate is shown in table 3.2.

3.3.5 Structure of a certificate on the domain server

Certificate on the domain server is similar in structure to the certificate stored in the contract. However, certificates for public subdomains and certificates for private subdomains are stored separately for the security purpose, as putting them together could reveal the private information to the public subdomain users. Structures of these certificates are shown in table 3.3 and table 3.4. Certificates of private subdomains have two additional fields: 1) Commitment to the subdomain name *privsd_comm* 2) Decommitment randomness *decom_rand* needed to open the commitment. Domain owner can choose to keep all the private subdomain certificates together or separate as per her security requirements. Both the certificates are signed by the domain owner's signing key *sk_owner* mentioned in section 3.3.6.

3.3.6 Keys assosiated with a domain

Each domain owner has two pair of keys (pk_{owner}, sk_{owner}) and $(pk_{update}, sk_{update})$ associated with the domain. (pk_{owner}, sk_{owner}) is used for signing and validating the transactions for registering and transferring the domain. It is also used for signing the transactions for registering the digital certificate for the domain. All other transactions such as registering subdomains, updating domain/subdomain information, registering certificates for subdomains, and revocation of certificates for domains/subdomains are signed by sk_{update}. The domain owner can change the update key whenever required. There is one more pair of key (pk_{enc}, sk_{enc}) which is used for secured communication over SSL/TLS on Internet. pk_{enc} is stored in the domain's digital certificate in the contract *SmartPKI*.

3.4 Smart Contracts for SmartDNSPKI

Smart Contracts for SmartDNSPKI are divided into two parts: One implemented for managing DNS and the other which is implemented for managing digital certificates. Both the smart contracts have multiple functions which can be invoked through transactions. All the transactions sent to the blockchain are signed by the transaction sender unless otherwise stated. Essentially, the data sent in a transaction also contains the signature of the transaction sender which is used by the miners to validate the transaction. Execution of a contract code occurs only after the transaction sender is validated.

Domain Certificate			
Domain Name			
Field	**Use**		
pk	Key of the domain used for encrypted communication over SSL/TLS		
valid_from	Specify the start period of the validity of the certificate		
valid_to	Specify the end period of the validity of the certificate		
revoked	Specify the revocation status of the certificate		
Public SD Certificates		**Private SD Certificates**	
Subdomain Name (1)		**Subdomain Commitment (1)**	
Field	**Use**	**Field**	**Use**
registered	Confirm that the public subdomain has been registered	registered	Confirm that the private subdomain has been registered
revoked	Specify the revocation status of the subdomain certificate	revoked	Specify the revocation status of the subdomain certificate
. . .		. . .	
Subdomain Name (n)		**Subdomain Commitment (n)**	
Field	**Use**	**Field**	**Use**
registered	Confirm that the public subdomain has been registered	registered	Confirm that the private subdomain has been registered
revoked	Specify the revocation status of the subdomain certificate	revoked	Specify the revocation status of the subdomain certificate

Table 3.2: Structure of a Domain Certificate in the contract storage

Domain Certificate	
Domain Name	
Field	**Use**
pk	Key of the domain used for encrypted communication over SSL/TLS
valid_from	Specify the start period of the validity of the certificate
valid_to	Specify the end period of the validity of the certificate
revoked	Specify the revocation status of the certificate
Public Subdomain Certificates	
Public Subdomain Name (1)	
Field	**Use**
revoked	Specify the revocation status of the subdomain certificate
. . .	
Public Subdomain Name (n)	
Field	**Use**
revoked	Specify the revocation status of the subdomain certificate
Domain owner's signature	

Table 3.3: Structure of a Domain Certificate with Public Subdomains stored on the domain server

Domain Certificate	
Domain Name	
Field	**Use**
pk	Key of the domain used for encrypted communication over SSL/TLS
valid_from	Specify the start period of the validity of the certificate
valid_to	Specify the end period of the validity of the certificate
revoked	Specify the revocation status of the certificate
Private Subdomain Certificates	
Private Subdomain Name (1)	
Field	**Use**
revoked	Specify the revocation status of the private subdomain certificate
privsd_comm	Specify the name of the subdomain
decom_rand	Specify the decommitment randomness needed to open the commitment
. . .	
Private Subdomain Name (n)	
Field	**Use**
revoked	Specify the revocation status of the subdomain certificate
privsd_comm	Specify the name of the subdomain
decom_rand	Specify the decommitment randomness needed to open the commitment
Domain owner's signature	

Table 3.4: Structure of a Domain Certificate with Private Subdomains stored on the domain server

3.4.1 Modules for SmartDNS contract

These are the modules for registering and updating domains/subdomains to the contract storage, transferring a domain to another address, and fetching information about the domain and subdomains from the contract storage. All these modules are implemented in the contract ***SmartDNS***.

3.4.1.1 Domain Registration

All domain owners need to register their domain and subdomains to the contract ***SmartDNS***. To register a domain *domain_name*, domain owner sends a transaction *domainreg* with hash of the domain name *hash(domain_name)* and domain registration fee π_d as function arguments. The client uses *keccak256* algorithm for hashing the domain name and registration fee π_d has been set to "0.0001 ETH". This transaction invokes the function *reg_domain* in the contract which checks if the sender has sufficient balance to pay for the registration and the domain hash is not already stored in the mapping *domain_owner* of the contract which stores domain hash as the key and the transaction sender address as the value.

Domain Registration

```
procedure DomainRegistration
    if $domain\_owner[hash(domain\_name)] = NULL \quad \&\& \quad msg.value \geq \pi_d$ then
        $domain\_owner[hash(domain\_name)] \leftarrow msg.sender$
        $block.coinbase.transfer \leftarrow \pi_d$
        /*hash(domain_name) is stored in the contract storage*/
        return success
    else
        return failure
    end if
end procedure
```

3.4.1.2 Domain Registration Confirmation

After waiting for 10-12 confirmations of the block containing the *domainreg* transaction, the domain owner sends a registration confirmation transaction *regconf* with the actual domain name *domain_name* and the data corresponding to the domain name e.g. its IP address or nameserver *addr*, its validity information *valid_from* and *valid_to*, and the update address *update_addr* as function arguments. This transaction invokes the function *reg_conf* of the contract which computes the hash of the domain name sent in this transaction and checks if the address stored in the *domain_owner* mapping corresponding to this hash is same as the transaction sender. After all the checks in the function are validated, *domain_name*, its IP address or nameserver*addr*, its validity information *valid_from* and *valid_to*, current transaction sender as the *owner_addr*, and the *update_addr* are stored in another mapping *domain_info* with transaction sender address as the key.

Domain Registration Confirmation

```
procedure DomainRegistrationConfirmation
    $domain\_name\_hashed \leftarrow hash(domain\_name)$
    if $domain\_owner[domain\_name\_hashed)] = msg.sender$ then
        $domain\_info[domain\_name].addr \leftarrow addr$
        $domain\_info[domain\_name].valid\_from \leftarrow valid\_from$
        $domain\_info[domain\_name].valid\_to \leftarrow valid\_to$
        $domain\_info[domain\_name].owner\_addr \leftarrow msg.sender$
        $domain\_info[domain\_name].update\_addr \leftarrow update\_addr$
        /*addr, valid_from, valid_to, owner_addr, and update_addr are
        stored in the contract storage*/
        return success
    else
        return failure
    end if
end procedure
```

3.4.1.3 Update Address Change

Domain owner can change the update address *update_addr* by sending the transaction *updateaddrs* with domain name *domain_name*, and the new update address *update_addr_new* as function arguments. This transaction invokes the function *update_addrs* of the contract which updates the update address of the domain after checking that the transaction sender is the owner of the domain.

Change Update Address

```
procedure ChangeUpdateAddress
    if $domain\_info[domain\_name].owner\_addr = msg.sender$ then
        $domain\_info[domain\_name].update\_addr \leftarrow update\_addr\_new$
        /*update_addr is updated in the contract storage*/
        return success
    else
        return failure
    end if
end procedure
```

3.4.1.4 Domain Transfer

The domain owner can transfer the domain to another address by sending the transaction *domtransfer* with domain name *domain_name* and the new address *owner_new* as function arguments signed by both sk_{owner} and sk_{update}. This transaction invokes the function *transfer_domain* of the contract which transfers the ownership of the domain to the new address *owner_new* after checking that the transaction sender is the current owner of the address.

Domain Transfer

```
procedure DomainTransfer
    if $domain\_info[domain\_name].owner\_addr = msg.sender$ then
        $domain\_info[domain\_name].owner\_addr = owner\_new$
        /*Domain's owner_addr is updated in the contract storage*/
        return success
    else
        return failure
    end if
end procedure
```

3.4.1.5 Domain Information Update

Domain information can be updated by sending the transaction *domupdate* with domain name *domain_name*, new IP address or new nameserver *addr_new*, updated validity information *valid_from_new* and *valid_to_new* as function arguments. This transaction invokes function *update_domain* of the contract which checks if the transaction is signed by the update key sk_{update} of the domain. Upon success, domain information is updated in mapping *domain_info* in the contract storage.

Domain Information Update

```
procedure DomainInformationUpdate
    if $domain\_info[domain\_name].update\_addr = msg.sender$ then
        $domain\_info[domain\_name].addr \leftarrow addr$
        $domain\_info[domain\_name].valid\_from \leftarrow valid\_from$
        $domain\_info[domain\_name].valid\_to \leftarrow valid\_to$
        /*addr, valid_from and valid_to are updated in the contract
        storage*/
        return success
    else
        return failure
    end if
end procedure
```

3.4.1.6 Domain Information Retrieval

Domain information can be retrieved by calling the function *get_info_domain* of the contract with *domain_name* as its argument. This call returns all the details of the domain like *addr* and its validity information, *valid_from* and *valid_to*.

Domain Information Retrieval

```
procedure DomainInformationRetrieval
    return ($domain\_info[domain\_name].addr$,
        $domain\_info[domain\_name].valid\_from, domain\_info$
        $[domain\_name].valid\_to$)
end procedure
```

3.4.1.7 Public Subdomain Registration

A public subdomain can be registered for a domain by sending the transaction *pubsdreg* with domain name *domain_name*, subdomain name *pubsd_name*, and its IP address or nameserver *addr* as the function arguments. This transaction invokes the function *reg_pubsd* of the contract which checks that the transaction is signed by the update key sk_{update} of the domain. After all the checks are validated, public subdomain, along with all its details, is stored in the contract storage.

Public Subdomain Registration

```
procedure PublicSubdomainRegistration
    if $domain\_info[domain\_name].update\_addr = msg.sender$ then
        $domain\_info[domain\_name].pubsd[count\_pubsd].name \leftarrow pubsd\_name$
        $domain\_info[domain\_name].pubsd[count\_pubsd].addr \leftarrow addr$
        $count\_pubsd \leftarrow count\_pubsd + 1$
        /*$pubsd\_name$ and $addr$ are added to the $pubsd$ array in the Domain structure of the domain stored in the contract storage.*/
        return $success$
    else
        return $failure$
    end if
end procedure
```

3.4.1.8 Public Subdomain Update

A public subdomain can be updated by sending the transaction *pubsdupdate* with domain name *domain_name*, subdomain name *pubsd_name*, and its new IP address or nameserver *addr_new* as the function arguments. This transaction invokes the function *update_pubsd* of the contract which checks that the transaction is signed by the update key sk_{update} of the domain. After all the checks are validated, *addr_new* is stored in the contract storage.

Public Subdomain Update

```
procedure PublicSubdomainUpdate
    if $domain\_info[domain\_name].update\_addr = msg.sender$ then
        $domain\_info[domain\_name].pubsd[index_{pubsd\_name}].addr \leftarrow addr\_new$
        /*$addr\_new$ is stored in the contract storage*/
        return $success$
    else
        return $failure$
    end if
end procedure
```

3.4.1.9 Public Subdomain Information Retrieval

Public subdomain information can be retrieved by calling the function *get_info_pubsd* of the contract with *domain_name* and *pubsd_name* as its argument. This call returns all the details of the subdomain like its IP address or nameserver *addr* and its validity information, *valid_from* and *valid_to* which are same as that of the domain.

Public Subdomain Information Retrieval

```
procedure PublicSubdomainInformationRetrieval
    return ($domain\_info[domain\_name].pubsd[index_{pubsd\_name}].addr$,
        $domain\_info[domain\_name].valid\_from, domain\_info$
        $[domain\_name].valid\_to$)
end procedure
```

3.4.1.10 Private Subdomain Registration

A private subdomain can be registered for a domain by sending the transaction *privsdreg* with domain name *domain_name*, subdomain name commitment *privsd_comm*, and a private nameserver *pns.domain_name* as the function arguments. This transaction invokes the function *reg_privsd* of the contract which checks that the transaction is signed by the update key sk_{update} of the domain. After all the checks are validated, private subdomain commitment *privsd_comm* along with its details is stored in the contract storage.

Private Subdomain Registration

```
procedure PrivateSubdomainRegistration
    if $domain\_info[domain\_name].update\_addr = msg.sender$ then
        $domain\_info[domain\_name].privsd[count\_privsd].name \leftarrow$
        $privsd\_comm$
        $domain\_info[domain\_name].privsd[count\_privsd].addr \leftarrow$
        $ns.privsd\_comm$
        $count\_privsd \leftarrow count\_privsd + 1$
        /*privsd_comm and ns.domain_name are added to the privsd
        array in the Domain structure of the domain stored in the
        contract storage*/
        return success
    else
        return failure
    end if
end procedure
```

3.4.1.11 Private Subdomain Update

A private subdomain can be updated by sending the transaction *privsdupdate* with domain name *domain_name*, commitment to the subdomain name *privsd_comm*, and its new nameserver *ns.domain_name_new* as the function arguments. This transaction invokes the function *update_privsd* of the contract which checks that the transaction is signed by the update key sk_{update} of the domain. After all the checks are validated, *ns.domain_name_new* is stored in the contract storage.

Private Subdomain Update

```
procedure PrivateSubdomainUpdate
    if $domain\_info[domain\_name].update\_addr = msg.sender$ then
        $domain\_info[domain\_name].privsd[index_{privsd\_comm}].addr \leftarrow$
        $ns.domain\_name\_new$
        /*ns.domain_name_new is stored in the contract storage*/
        return success
    else
        return failure
    end if
end procedure
```

3.4.1.12 Private Subdomain Information Retrieval

Private subdomain information can be retrieved by calling the function *get_info_privsd* of the contract with *domain_name* and *privsd_comm* as its argument. This call returns all the details of the subdomain like its nameserver *ns.domain_name* and its validity information, *valid_from* and *valid_to* which are same as that of the domain.

Private Subdomain Information Retrieval

```
procedure PublicSubdomainInformationRetrieval
    return ($domain\_info[domain\_name].pubsd[index_{pubsd\_name}].addr,$
        $domain\_info[domain\_name].valid\_from, domain\_info$
        $[domain\_name].valid\_to$)
end procedure
```

3.4.2 Modules for SmartPKI contract

These are the modules for registering, updating, revoking, and fetching the certificates for a domain as well as for its subdomains. All these modules are implemented in the contract ***SmartPKI***. This contract interacts with the contract ***SmartDNS*** for verifying the domain owner and to fetch the validity information of the domain.

3.4.2.1 Certificate Registration

After the domain name has been successfully registered in the contract *SmartDNS*, domain owner can register a certificate for the domain by sending the transaction *certreg* signed by the key sk_{owner} with the domain name *domain_name*, domain's public key pk_{domain_name}, certificate's validity information *valid_from*, and certificate registration fee π_c which is set to value "0.0001 ETH", as the function arguments. This transaction invokes the function *reg_cert* of the contract which checks that no certificate has been registered for the domain earlier or the registered certificate has been revoked. After the checks are validated, certificate for the domain is stored in the contract storage.

Certificate Registration

```
procedure CertificateRegistration
    if $SmartDNS.domain\_info[domain\_name].owner\_addr = msg.sender$
        && $msg.value \geq \pi_c$ then
        if $certificate[domain\_name] = ``''$ ||
        $certificate[domain\_name].revoked = true$ then
            $certificate[domain\_name].pk \leftarrow pk_{domain\_name}$
            $certificate[domain\_name].valid\_from \leftarrow valid\_from$
            $certificate[domain\_name].valid\_to \leftarrow$
            $SmartDNS[domain\_info].valid\_to$
            $certificate[domain\_name].revoked \leftarrow false$
            $block.coinbase.transfer \leftarrow \pi_c$
            /* Certificate($pk_{domain\_name}$, valid_from, valid_to,
            revocation status) is stored in the contract storage*/
            return success
        else
            return failure
        end if
    else
        return failure
    end if
end procedure
```

3.4.2.2 Certificate Update

Certificate for a domain can be updated by sending the transaction *certupdate* with the domain name *domain_name*, the updated public key pk_{domain_name}*-new*, and the updated validity information *valid_from_new* as the function arguments. Certificate's *valid_to* field is same as that of the domain's *valid_to* field in the *SmartDNS* contract. This transaction invokes the function *update_cert* of the contract which updates the certificate information in the contract storage after verifying that the transaction is signed by the key sk_{update} of the domain *domain_name*.

Certificate Update

```
procedure CertificateUpdate
    if $SmartDNS.domain\_info[domain\_name].update\_addr = msg.sender$
    then
        if $certificate[domain\_name].valid\_from > 0$   &&
        $certificate[domain\_name].revoked = false$ then
            $certificate[domain\_name].pk \leftarrow pk_{domain\_name}\text{-}new$
            $certificate[domain\_name].valid\_from \leftarrow valid\_from\_new$
            $certificate[domain\_name].valid\_to \leftarrow$
            $SmartDNS[domain\_info].valid\_to$
            $certificate[domain\_name].revoked \leftarrow false$
            /* Certificate($pk_{domain\_name}$-new, valid_from_new, valid_to,
            revocation status) is stored in the contract storage*/
            return success
        else
            return failure
        end if
    else
        return failure
    end if
end procedure
```

3.4.2.3 Certificate Revocation

Certificate for a domain can be revoked by sending the *certrevoke* transaction with the domain name *domain_name* as the function argument. This transaction calls the *revoke_cert* function of the contract which sets the revocation status *revoked* of the domain certificate *true* after verifying that the transaction is signed by the key sk_{update} of the domain *domain_name*.

Certificate Revocation

```
procedure CertificateRevocation
    if $SmartDNS.domain\_info[domain\_name].update\_addr = msg.sender$
        then
        $certificate[domain\_name].revoked \leftarrow true$
        /* Certificate's revocation status *revoke* is set true*/
        return *success*
    else
        return *failure*
    end if
end procedure
```

3.4.2.4 Domain Certificate Retrieval

Domain certificate can be retrieved by calling the function *get_cert* of the contract with *domain_name* as its argument. This call returns all the details stored in the domain's certificate i.e its public key pk_{domain_name}, its validity information, *valid_from* and *valid_to*, and its revocation status *revoked*.

Domain Certificate Retrieval

```
procedure DomainCertificateRetrieval
    return $(certificate[domain\_name].pk, certificate[domain\_name].$
        $valid\_from, certificate[domain\_name].valid\_to, certificate$
        $[domain\_name].revoked)$
end procedure
```

3.4.2.5 Public Subdomain Certificate Registration

Certificate for a public subdomain can be registered by sending the transaction *pubsdcertreg* with domain name *domain_name* and the public subdomain name *pubsd_name* as the arguments for the function. This transaction invokes the function *reg_cert_pubsd* of the contract which stores the certificate of the public subdomain after checking that the transaction is signed by the key sk_{update} of the domain *domain_name*.

Public Subdomain Certificate Registration

```
procedure PublicSubdomainCertificateRegistration
    if $SmartDNS.domain\_info[domain\_name].update\_addr = msg.sender$
    then
        $certificate[domain\_name].public\_sd[pubsd\_name].registered \leftarrow true$
        $certificate[domain\_name].public\_sd[pubsd\_name].revoked \leftarrow false$
        /* Certificate_SD(true, false) is stored in the contract storage*/
        return success
    else
        return failure
    end if
end procedure
```

3.4.2.6 Public Subdomain Certificate Revocation

Certificate for a public subdomain can be revoked by sending the *pubsdcertrevoke* transaction with the domain name *domain_name* and its public subdomain name *pubsd_name* as the function arguments. This transaction calls the *revoke_cert_pubsd* function of the contract which sets the revocation status *revoked* of the subdomain certificate *true* after verifying that the transaction is signed by the key sk_{update} of the domain *domain_name*.

Public Subdomain Certificate Revocation

```
procedure PublicSubdomainCertificateRevocation
    if $SmartDNS.domain\_info[domain\_name].update\_addr = msg.sender$
    then
        $certificate[domain\_name].public\_sd[pubsd\_name].revoked \leftarrow true$
        /* Public Subdomain Certificate's revocation status revoke is set true*/
        return success
    else
        return failure
    end if
end procedure
```

3.4.2.7 Public Subdomain Certificate Retrieval

Public Subdomain certificate can be retrieved by calling the function *get_cert_pubsd* of the contract with *domain_name* and the subdomain name *pubsd_name* as its arguments. This call returns all the details stored in the subdomain's certificate i.e its public key pk_{domain_name}, its validity information, *valid_from* and *valid_to*, and its revocation status *revoked*.

Public Subdomain Certificate Retrieval

```
procedure PublicSubdomainCertificateRetrieval
    return ($certificate[domain\_name].pk, certificate[domain\_name].$
    $valid\_from, certificate[domain\_name].valid\_to, certificate$
    $[domain\_name].revoked \quad || \quad certificate[domain\_name].public\_sd$
    $[pubsd\_name].revoked$)
end procedure
```

3.4.2.8 Private Subdomain Certificate Registration

Certificate for a private subdomain can be registered by sending the transaction *privsdcertreg* with domain name *domain_name* and the private subdomain name commitment *privsd_comm* as the arguments for the function. The private subdomain name *priv_sd* and the decommitment randomness *r* is stored in the certificate in the domain server which is presented to the client when requested for. The transaction *privsdcertreg* invokes the function *reg_cert_privsd* of the contract which stores the certificate of the private subdomain after checking that the transaction is signed by the key sk_{update} of the domain *domain_name*.

Private Subdomain Certificate Registration

```
procedure PrivateSubdomainCertificateRegistration
    if $SmartDNS.domain\_info[domain\_name].update\_addr = msg.sender$
        then
         $certificate[domain\_name].private\_sd[privsd\_comm].registered$
         $\leftarrow true$
         $certificate[domain\_name].private\_sd[privsd\_comm].revoked \leftarrow$
        $false$
         /* Certificate_SD(true, false) is stored in the contract storage*/
         return success
    else
         return failure
    end if
end procedure
```

3.4.2.9 Private Subdomain Certificate Revocation

Certificate for a private subdomain can be revoked by sending the *privsdcertrevoke* transaction with the domain name *domain_name* and its private subdomain name commitment *pubsd_name* as the function arguments. This transaction calls the *revoke_cert_pubsd* function of the contract which sets the revocation status *revoked* of the subdomain certificate *true* after verifying that the transaction is signed by the key sk_{update} of the domain *domain_name*.

Private Subdomain Certificate Revocation

```
procedure PrivateSubdomainCertificateRevocation
    if $SmartDNS.domain\_info[domain\_name].update\_addr = msg.sender$
    then
        $certificate[domain\_name].private\_sd[pubsd\_name].revoked \leftarrow true$
        /* Private Subdomain Certificate's revocation status revoke is set
        true*/
        return success
    else
        return failure
    end if
end procedure
```

3.4.2.10 Private Subdomain Certificate Retrieval

Private Subdomain certificate can be retrieved by calling the function *get_cert_privsd* of the contract with *domain_name* and the commitment to the private subdomain name *privsd_comm* as its arguments. This call returns all the details stored in the subdomain's certificate i.e its public key pk_{domain_name}, its validity information, *valid_from* and *valid_to*, and its revocation status *revoked*.

Private Subdomain Certificate Retrieval

```
procedure PublicSubdomainCertificateRetrieval
    return ($certificate[domain\_name].pk, certificate[domain\_name].$
    $valid\_from, certificate[domain\_name].valid\_to, certificate$
    $[domain\_name].revoked \quad || \quad certificate[domain\_name].$
    $private\_sd[privsd\_comm].revoked$)
end procedure
```

3.5 Certificate validation during Client-Server interaction

Case 1: When a client requests for the domain's/public subdomain's certificate, the server presents the certificate in table 3.3. Client first fetches the address of the domain owner *owner_addr* from *SmartDNS* contract and validates the domain owner's signature present on the certificate. It then fetches the certificate from the *SmartPKI* contract and compares it with the certificate presented by the server. If they match and the certificate is not expired or revoked, it extracts the public key from the certificate and uses it for secure communication with the server.

Case 2: When a client requests for a private subdomain's certificate, the server presents the certificate in table 3.4. Client first validates the signature present on the certificate as explained in case 1. It then extracts the commitment to the private subdomain *privsd_comm* and the decommitment randomness *decom_rand* from the certificate and validates the commitment. Next, it gets the certificate corresponding to the commitment from the domain's certificate in the contract *SmartPKI* and compares it with the certificate presented by the server. If they match and the certificate is not expired or revoked, it proceeds as in Case 1.

3.6 Security of the scheme

Security of a smart contract is dependent on the security of the underlying blockchain on which it has been implemented. We have chosen Ethereum as our blockchain for the ease of use. Therefore, security of our scheme is dependent on the security of Ethereum. As long as a considerable number of Ethereum miners do not go rogue, the scheme implemented on the blockchain is assumed to be secure if there are no implementation glitches in the contracts. In our proposal, individual domains registered in the contract *SmartDNS*, and their corresponding certificates registered in the contract *SmartPKI* are secure as long as the keys associated with the accounts responsible for their registration, update, and revocation are secure.

We attached three pairs of keys (pk_{owner}, sk_{owner}), $(pk_{update}, sk_{update})$, and (pk_{enc}, sk_{enc}) with a domain owner with their respective functionalities discussed in section 3.3.6. We can implement offline signing of all the transactions to protect the key pairs (pk_{owner}, sk_{owner}), $(pk_{update}, sk_{update})$ from online attacks. However, the key (pk_{enc}, sk_{enc}) is susceptible to online attacks. We consider, and discuss all the possibilities of key losses and how a user can overcome these situations below. We discuss the security in two parts: *1) Unsigned key updates:* these updates do not require signature of the key to be updated, *2) Signed key updates:* these updates do require signature of the key to be updated.

3.6.1 Unsigned key updates

These updates give the user ease of reclaiming the ownership of the lost keys in case the keys are stolen. However, these updates are less secure compared to the signed key updates.

Case 1: Only sk_{enc} is compromised
The key pair *(pk_{enc}, sk_{enc})* can be updated by sending the transaction *certupdate* with the new public key *pk_{enc}_new* signed by the update key sk_{update}.
Case 2: Only sk_{update} is compromised
Domain owner can always change the update key pair *(pk_{update}, sk_{update})* by sending *updateaddrs* transaction as described in section 3.4.1.3.
Case 3: Both sk_{enc} and sk_{update} are compromised
Domain owner can first change the update key pair *(pk_{update}, sk_{update})* as discussed in Case 2. Then she can change *(pk_{enc}, sk_{enc})* as discussed in Case 1.

All other cases of key loss which include compromise of sk_{owner} cannot be handled by unsigned key updates.

3.6.2 Signed key updates

Signed key updates in our proposal are essentially multisig schemes which require signatures from multiple signing keys (at least 2, or all the 3 keys described in section 3.3.6) for any kind of key update. These updates give more security to the user at the cost of increased complexity. Moreover, compromised keys cannot be reclaimed if the keys are stolen.

Case 1: Only sk_{enc} is compromised
The compromised key pair *(pk_{enc}, sk_{enc})* can be updated by sending the transaction *certupdate* with the new public key *pk_{enc}_new* signed by both sk_{enc} and the update key sk_{update}.
Case 2: Only sk_{update} is compromised
Domain owner can change the update key pair *(pk_{update}, sk_{update})* by sending *updateaddrs* transaction as described in section 3.4.1.3 additionally signed by the sk_{update}.
Case 3: Both sk_{enc} and sk_{update} are compromised
Domain owner can first change the update key pair *(pk_{update}, sk_{update})* as discussed in Case 2. Then she can change *(pk_{enc}, sk_{enc})* as discussed in Case 1.
Case 4: Only sk_{owner} is compromised
Domain owner can transfer the domain to another address by sending the transaction *domtransfer* as discussed in section 3.4.1.4.
Case 5: Both sk_{owner} and sk_{update} are compromised
Domain owner can still regain the sole ownership of the domain by sending the transaction *domtransfer* as discussed in section 3.4.1.4 with an additional signature by sk_{enc}.

Case where all the three keys are compromised cannot be handled by this scheme, as of now.

Chapter 4

Implementation

In this chapter, we briefly talk about Ethereum, the blockchain-based platform on which we have implemented our scheme, and the Smart contracts, followed by the implementation and performance details of the scheme.

4.1 Ethereum

Ethereum [15, 26] was proposed by Vitalik Buterin, a cryptocurrency researcher and a programmer involved with Bitcoin, in a white paper [15] in late 2013 with a goal of building decentralised applications. Ethereum is an open-source, public, blockchain-based distributed computing platform featuring smart contract (scripting) functionality, which facilitates online contractual agreements. It provides a decentralised Turing-complete virtual machine, the Ethereum Virtual Machine (EVM), which can execute scripts using an international network of public nodes. Ethereum also provides a cryptocurrency token called "ether", which can be transferred between accounts and used to compensate participant nodes for computations performed. Gas, an internal transaction pricing mechanism, is used to mitigate spam and allocate resources on the network.

In Ethereum, the state is made up of objects called "accounts", with each account having a 20-byte address and state transitions being direct transfers of value and information between accounts. In general, there are two types of accounts: externally owned accounts, controlled by private keys, and contract accounts, controlled by their contract code. Externally owned account interact with blockchains by sending signed data packages called "transactions"that store messages to be sent to the blockchain. Contracts interact with each other through "messages". The state of the blockchain is stored in a tree data structure called "Merkle-Patricia"tree.

4.2 Smart contracts

A smart contract is a computerized transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs. In Ethereum, smart contracts are treated as autonomous scripts or stateful decentralised applications that are stored in the Ethereum blockchain for later execution by the EVM. Instructions embedded in Ethereum contracts are paid for in ether (or more technically "gas") and can be implemented in a variety of Turing complete scripting languages such as Solidity, Serpent, LLL, etc. which are complied down to low-level, stack-based bytecode language referred to as "EVM code"for execution.

Our main proposal is to use the framework of Ethereum for creating and managing smart contract based, privacy-preserving, DNS and PKI, where domain owners can register their domain, corresponding public and private subdomains along with a digital certificate for all her domains and subdomains from one externally-owned account. Our scheme also provides all the functionalities of a PKI like registration, update, retrieval, and revocation of digital certificates through smart contracts. All the information related to a domain and its certificate is stored on the blockchain.

4.3 Implementation

All the algorithms have been implemented in two contracts: *SmartDNS* and *SmartPKI* in 151 and 105 lines of solidity respectively.

A corresponding command-line Python client has been implemented for carrying out all the transactions for both the contracts. The client allows a user to register and update all the domains as well as public and private subdomains. The client uses *keccak256* algorithm for hashing the domain name in the first step of the domain registration step and *Pedersen commitment scheme* for computing the commitment to the private subdomains. Domain registration fee and certificate registration fee is transferred to the miner's account which ensures faster mining of these transactions by the miners. Contracts have been tested on the testnet on Ethereum Wallet. Ethereum Wallet is a framework for storing and transferring Ether as well as writing, deploying, and testing smart contracts without the need of real ethers.

4.4 Performance

Gas costs of all the transactions have been analyzed using the implementation on the testnet. Costs associated with all the transactions of contracts *SmartDNS*

SmartDNS				
Transaction	**Data** (Bytes)	**Gas**	**Ether**	**USD($)**
contract deployment	13614	3145988	0.056627784	15.969035088
domainreg	36	44807	0.000806526 + 0.0001(domain registration fee)	0.255640332
regconf	292	130122	0.002342196	0.660499272
domupdate	292	54810	0.00098658	0.27821556
domtransfer	132	31455	0.00056619	0.15966558
updateaddrs	132	31568	0.000568224	0.160239168
pubsdreg	292	90259	0.001624662	0.458154648
pubsdupdate	292	50918	0.000916524	0.258459768
privsdreg	292	75217	0.001353906	0.381801492
privsdupdate	292	51,693	0.000930474	0.262393668

Table 4.1: Gas costs of all the transactions in SmartDNS contract

and *SmartPKI* are compiled in table 4.1 and 4.2 respectively. Gas price is fixed at 1 gas = 0.000000018 ether and ether to USD conversion rate is 1 ether = 282 USD. Implementation cost is one-time cost incurred during deployment of the contracts. The costs presented in the tables may go higher during real-time implementation of the scheme and will vary with the amount of data sent in the transactions. However, these costs are still much less compared to current domain registration and renewal fees in DNS and certificate registration fees charged by Certificate Authorities.

SmartPKI				
Transaction	**Data** (Bytes)	**Gas**	**Ether**	**USD($)**
contract deployment	11355	2686552	0.048357936	13.636937952
certreg	356	207615	0.00373707 + 0.0001(certificate registration fee)	1.08205374
certupdate	356	87895	0.00158211	0.44615502
certrevoke	100	32806	0.000860508	0.242663256
pubsdcertreg	196	50350	0.0009063	0.2555776
pubsdcertrevoke	196	36200	0.0006516	0.1837512
privsdcertreg	196	50480	0.00090864	0.25623648
privsdcertrevoke	196	36198	0.000651564	0.183741048

Table 4.2: Gas costs of all the transactions in SmartPKI contract

Chapter 5

Future work and Conclusion

We have proposed and implemented a prototype for a decentralized, blockchain-based, privacy-preserving scheme for Domain Name System and Public Key Infrastructure, which supports functionalities such as domain registration, update and transfer, and certificate registration, update, and revocation. Our scheme supports the registration of private subdomains and their certificates in a privacy-preserving fashion. In addition to that, it provides linkability between the DNS and the PKI mandating the domain owner and the certificate owner of the domain to be the same.

Our future goal is to integrate this scheme with distributed databases such as IPFS [12] in order to lower the amount of data getting stored directly in the blockchain. We also plan to implement this scheme in entirety and gather real-time performance analysis, and measure its scalability, which is a measure challenge for the blockchain-based applications.

Bibliography

[1] Certificate authority. https://en.wikipedia.org/wiki/Certificate_authority.

[2] Certificate transparency: Domain label redaction. https://tools.ietf.org/id/draft-strad-trans-redaction-01.html.

[3] Diginotar. https://en.wikipedia.org/wiki/DigiNotar.

[4] Domain name. https://en.wikipedia.org/wiki/Domain_name.

[5] Namecoin. https://namecoin.org/.

[6] Pki. https://en.wikipedia.org/wiki/Public_key_infrastructure.

[7] Trustwave issued a man-in-the-middle certificate. http://www.h-online.com/security/news/item/Trustwave-issued-a-man-in-the-middle-certificate-1429982.html.

[8] Unauthentic ”microsoft corporation” certificates. https://www.cert.org/historical/advisories/CA-2001-04.cfm.

[9] Web of trust. https://en.wikipedia.org/wiki/Web_of_trust.

[10] Mustafa Al-Bassam. Scpki: A smart contract-based pki and identity system. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, pages 35–40. ACM, 2017.

[11] Muneeb Ali, Jude C Nelson, Ryan Shea, and Michael J Freedman. Blockstack: A global naming and storage system secured by blockchains. In *USENIX Annual Technical Conference*, pages 181–194, 2016.

[12] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[13] Joseph Bonneau. Ethiks: Using ethereum to audit a coniks key transparency log. In *Financial Cryptography Workshops*, pages 95–105, 2016.

[14] Tom Brewster. Diginotar goes bankrupt after hack, 2011. http://www.itpro.co.uk/636244/diginotar-goes-bankrupt-after-hack.

[15] Vitalik Buterin et al. Ethereum white paper, 2013.

[16] Germano Caronni. Walking the web of trust. In *IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises - WETICE 2000*, pages 153–158, 2000.

[17] Saba Eskandarian, Eran Messeri, Joe Bonneau, and Dan Boneh. Certificate transparency with privacy. *arXiv preprint arXiv:1703.02209*, 2017.

[18] Conner Fromknecht, Dragos Velicanu, and Sophia Yakoubov. CertCoin: A NameCoin based decentralized authentication system. Technical report, Massachusetts Institute of Technology, MA, USA. 6.857 Class Project, 2014. `https://courses.csail.mit.edu/6.857/2014/files/19-fromknecht-velicann-yakoubov-certcoin.pdf`.

[19] Harry A Kalodner, Miles Carlsten, Paul Ellenbogen, Joseph Bonneau, and Arvind Narayanan. An empirical study of namecoin and lessons for decentralized namespace design. In *WEIS*, 2015.

[20] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency, June 2013.

[21] Stephanos Matsumoto and Raphael M Reischuk. Ikp: Turning a pki around with decentralized automated incentives. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 410–426. IEEE, 2017.

[22] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. Coniks: Bringing key transparency to end users. In *USENIX Security Symposium*, pages 383–398, 2015.

[23] RL'Bob' Morgan, Kurt D Zeilenga, Jeff Hodges, and Peter Saint-Andre. Best practices for checking of server identities in the context of transport layer security (tls). 2011.

[24] Jon Postel. Domain name system structure and delegation. 1994.

[25] Brian Prince. Comodo attack sparks ssl certificate security discussions, 2011. `http://www.crn.com/news/security/229400284/comodo-attack-sparks-ssl-certificate-security-discussions.htm`.

[26] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.