Rethinking Certificate Authorities: Understanding and decentralizing domain validation

Seyedehmahsa Moosavi

A thesis in the

Concordia Institute for Information Systems Engineering

Presented in Partial Fulfilment of the Requirements for

the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

April 2, 2018

CONCORDIA UNIVERSITY

Division of Graduate Studies

This is to certify that the thesis prepared

By : **Seyedehmahsa Moosavi**

Entitled : **Rethinking Certificate Authorities: Understanding and decentralizing domain validation**

and submitted in partial fulfilment of the requirements for the degree of

Master of Applied Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee :

______________________________ Chair
Dr. W. Lucia

______________________________ Supervisor
Dr. Jeremy Clark

______________________________ CIISE Examiner
Dr. A. Youssef

______________________________ External Examiner
Dr. F. McKelvey

Approved by ______________________________
Dr. C. Assi
Graduate Program Director

______________ 2018.

____
Dr. A. Asif
Dean of ENCS

ABSTRACT

Rethinking Certificate Authorities:

Understanding and decentralizing domain validation

Seyedehmahsa Moosavi

HTTPS (HTTP over TLS) protocol provides message integrity, confidentiality, and server authentication. Server authentication relies on the client's ability to obtain a correct public key which is bound to the server. To provide this, the Public Key Infrastructure (PKI) uses a system of trusted third parties (TTPs) called the certificate authorities (CAs). CAs are the companies who receive certificate requests for domain names, they then use validation techniques to verify the ownership of those domains and once verified, they issue the digital certificates. These digital certificates are the electronic documents which simply bind domain names to the cryptographic keys and can be further used to secure communication channels over the web. However, PKI's several drawbacks enabled the malicious parties to break the entire CA model and issue themselves fraudulent certificates for domain names.

There has been little quantitative analysis of the certificate authorities (CAs) and how they establish domain names validation, so we first perform a thorough empirical study on the CA ecosystem and evaluate the security issues with the domain verification techniques. We find out that a central problem with the certificate model is that CAs resort to *indirection* to issue certificates because they are not directly authoritative over who owns what domain. Therefore, we design and implement a new and useful paradigm for thinking about who is actually authoritative over PKI information in the web certificate model. We then consider what smart contracts could add to the web certificate model, if we move beyond using a blockchain as passive, immutable (subject to consensus) store of data. To illustrate the potential, we develop and experiment with an Ethereum-based web certificate model we call Ghazal*, discuss different design decisions, and analyze deployment costs.

Acknowledgments

First and foremost, I would like to express my heartfelt gratitude to my supervisor, Dr. Jeremy Clark, without him I could not have completed my Master's degree. I am always indebted for his continuous support, encouragement, patience, friendship, and immense knowledge. He has taught me how to do research and guided me throughout my studies at Concordia University. Simply put, you are the best supervisor and advisor I could have ever asked for.

My special thanks to my office mates and colleagues for making my experience at Concordia University pleasant. I am especially thankful to Shayan Eskandari, for his support and help he has provided.

I would like to thank my parents and little brother for being loving and caring throughout my whole life. I left you behind to start a new journey in Canada and I simply miss you in every single moments of my life. I extremely appreciate your kindness, love, and support in all stages of my life.

Last but not least, I want to deeply thank a person who means a lot to me, my husband and my best friend, Kaveh, without whose love and support I just could not have been "myself". I cried with you, I laughed with you, and I grew up with you. Thank you for always being there for me and for believing in me all the time.

Mahsa Moosavi

Contents

List of Figures

Chapter 1

Introductory Remarks

In the recent years, the Internet has gained a significant ground on facilitating world-wide communications. Unlike the past when Internet was only used as a medium for sharing files, it is now the principle of everyone's life. Communications take place over the web using unlimited number of electronic devices. Thus, individuals need to be able to authenticate the digital identities of remote parties to establish secure web communication channels. These digital identities are managed by a group of third parties such as Certificate Authorities (CAs), ICANN, and DNS. However, placing too much trust on these authorities has led to single point of failures in the entire web.

Given the centralized design of the web entities (*i.e.,* DNS, ICANN, CAs), studies have considered blockchain-approaches to manage digital identities in a decentralized manner (using a consensus mechanism) to roughly eliminate the role of trusted third parties in providing secure and authenticated transmissions of data over the web.

In this thesis, we would like to look at the public key infrastructure (PKI) and certificate authorities (CAs) who manage digital identities on the web. We particularly investigate the identity validation methods applied by the real-world CAs. We also reevaluate the concept of *authority* — what does it mean to be authoritative over something?, and introduce the uni-authoritative paradigm. Eventually, we represent our system Ghazal*, a smart contract-based naming and PKI uni-authoritative system which is implemented and tested on the Ethereum blockchain.

1.1 Background

The HTTPS (HTTP over SSL/TLS) protocol enables secure connections to websites with confidentiality, message integrity, and server authentication. Server authentication relies on a client being able to determine the correct public key for a server. HTTPS protocol uses the public key infrastructure (PKI) which refers to a set of methods which enables the creation, distribution, usage of public key certificates for a network of users. Certificate authorities (CAs) are the main components of PKI that provide the *(identity, public key)* binding in the form of a certificate. Client devices, through the browser and/or the operating system, are pre-installed with a set of known CAs who can delegate their authority to intermediary CAs through a protocol involving certificates. When a CA issues a certificate to a web-server, there are generally three types: domain validated (DV) certificates bind a public key only to a domain (*e.g.,* `example.com`), while organization validated (OV) and extended validated (EV) certificates validate additional information about the organization that operates the server (Example, Inc.). By issuing certificates, CAs (as trusted third parties) certify that specific public key belongs to a certain subject who owns the corresponding private key. Thus, when a client visits a website, he can verify the server's public key based on his or her identity [2, 3, 4, 5, 6, 7].

The current web certificate design has led the security of the web to depend on anchoring trust in CAs — the trust appears once CAs assure the security of their private signing key. Despite CA's solid effort and operation, there have been quite a few incidents in which a CA's private keys were compromised as a result of social engineering attacks, governmental obligation, functional mistakes *etc.* As an example, in 2011, Comodo[8, 9] and DigiNotar [10], two of the most significant CAs on the web, have been compromised. Other failure occurred in other certificate authorities such as Turkish and French CAs who issued unauthorized certificates for several Google domains in 2013 [11, 12]. In some cases, failures and misbehaviours of certificate authorities resulted in several attacks such as *man in the middle (MITM)* attacks against popular domains. For instance, attackers were successfully able to issue a fraudulent digital certificate for extremely high-value domains including `google.com` and `login.yahoo.com` and simply intercepted communications with these well-known sites [13]. Compromise of CAs private keys introduces single point-of-failures in the entire PKI and results in large number of security violations on the web.

Therefore, one application of blockchain technology that has received some research and commercial interest is the idea of replacing (or augmenting) the web certificate model used by clients (OS and browsers) to form secure communication channels with web-servers. This model has been plagued with issues from fraudulent certificates used to impersonate servers to ineffective revocation mechanisms; see Clark and van Oorschot for a survey [14].

Namecoin is an altcoin (software based on Bitcoin with a distinct blockchain) that implements a decentralized namespace for domain names[15]. The main feature of Namecoin is that for a fee, users can register a `.bit` address and map it to an IP address of their choice. CertCoin [16], and PB-PKI [17] are extensions to Namecoin that add the ability to specify an HTTPS public key certificate for the domain (as well as other PKI operations like expiration and revocation, which we discuss in Section 4.2.1.4). Blockstack [18] achieves the same goal by embedding data into a root blockchain, a process called virtualchains that could be instantiated with `OP_RETURN` on Bitcoin's blockchain.

Some research has looked at adding transparency, effectively through an efficient log of CA-issued certificates, to augment the current web certificate model. This is a very active area of research that includes certificate transparency (CT) [19], sovereign keys (SKs) [20], and ARPKI [21]. IKP [22] provides an Ethereum-based system for servers to advertise policies about their certificates (akin to a more verbose CAA on a blockchain instead via DNS). Research a bit further removed from web certificates concerns decentralized PKIs and broader identities. While not decentralized, CONIKS provides a distributed transparency log similar to CT but for public keys (while they could be for anything, email and IM are the primary motivations) [23]. Bonneau provides an Ethereum smart contract for monitoring CONIKS [24]. ClaimChain is similar to CONIKS but finds a middle-ground between a small set of distributed servers (CONIKS) and a fully decentralized but global state (blockchain) by having fully decentralized, local states that can be cross-validated [25]. CONIKS and ClaimChain do not use CAs but rather rely on users validating the logs, which are carefully designed to be non-equivocating. ChainAnchor provides identity and access management for private blockchains [26], while CoSi is a distributed signing authority generic logging [27]. Each of these systems is concerned with logging data (a generic umbrella that encapsulates many of these is Transparency Overlays [28]).

Finally, some research has explored having public validated by external parties but replacing the role of CAs with a PGP-style web of trust. SCPKI is an implementation of this idea on Ethereum [29]. Our observation is that for domain validation, a blockchain with a built-in naming system is already authoritative over the namespace and does not require additional validation.

1.2 Contributions

The primary contributions of our work are as follows:

- **Empirical study of certificate validation methods.** We perform a thorough empirical study of certificate authorities and the validation techniques they employ. We develop a full enumeration of all uniquely identified trusted CAs in the real world. In total, we make an in-depth investigation of ∼700 certificate authorities and provide an overview of the validation methods they rely on to issue certificates.

- **Rethinking authorities in the PKI.** Given the fundamental issues with the current web certificate model, we reevaluate the concept of *authority* — what does it mean to be authoritative over something? We claim that certificate authorities are not any more or less authoritative over who owns what domain than you or I.

- **Decentralized PKI for the web.** We design and implement Ghazal*, a smart contract-based naming and PKI uni-authoritative system. This system is built and tested on the Ethereum blockchain.

1.3 Organization

The rest of the document is organized in the following way:

- In Chapter 2, we provide an overview of the background information about cryptography, public key certificates along with its components, Blockchain technology, and Ethereum blockchain.

- In Chapter 3, we provide an empirical study of the certificate authorities and the identity validation methods they use to issue certificates. We then pinpoint the fundamental issues with the current web certificate model.

- In Chapter 4, we provide the detailed introduction of our proposed system Ghazal* along with its design decisions and functions.

- In Chapter 5, we provide in-depth real-world implementation and evaluation details of our system Ghazal*.

- In Chapter 6, we provide conclusion and a brief discussion for this dissertation.

Chapter 2

Preliminaries and Related Work

This chapter is divided into four main sections, each of which provides a brief overview of the main concepts that a reader without sufficient background needs before reading the thesis.

2.1 Cryptography

Cryptography is the application of techniques used to provide secure communication channels where message confidentiality and integrity are assured. This section discusses a few basic concepts in cryptography.

2.1.1 Public Key Encryption

Public key cryptography is an encryption scheme [30] where each user owns a pair of keys: (1) a public key Pk, which is known to everybody and (2) a private key Sk, which is a secret key. Any user can encrypt a plain-text message P using the recipient's public key Pk, and the cipher-text message C can only be decrypted using the corresponding private key. This encryption technique uses a cryptographic algorithm A_e for encryption and A_d for decryption. Equations 2.1 and 2.2 represents the encryption and decryption in public key encryption scheme.

$$C = A_{e,Pk}(M) \tag{2.1}$$

$$M = A_{d,Sk}(C) \tag{2.2}$$

2.1.2 Digital Signatures

A digital signature σ resembles a physical signature which proves the authenticity of messages on the web [31]. When *Alice* digitally signs the message m using her private key Sk, anybody on the web can verify the signature using the corresponding public key Pk to assure m effectively belongs to *Alice* as well as it has not been altered while being transferred. Digital signatures are part of the public key cryptography scheme — signing a message is equal to encrypting it under the private key. Equations 2.3 and 2.4 represent creation and verification of digital signatures.

$$\sigma = \mathsf{Sign}_{sk}(m) \tag{2.3}$$

$$\texttt{T/F} = \mathsf{Verify}_{pk}(m, \sigma) \tag{2.4}$$

2.1.3 Digital Hash Functions

A cryptographic hash function $\mathcal{H}(m)$ creates fixed size length outputs called *hash values* for any arbitrary size inputs (*pre-image*)[32]. These functions are used to verify whether a candidate pre-image is equal to the real pre-image value. A perfect cryptographic hash function is non-invertible, meaning that it is infeasible to generate a pre-image from its hash value, this property is referred to as the *pre-image resistance*.

Another property of an ideal hash function is called *collision resistance*, that is, it should be infeasible to find two values x and y in such a way that $\mathcal{H}(x) = \mathcal{H}(y)$, and $x \neq y$.

2.2 Public Key Infrastructure (PKI)

As it is described in PKIX IETF Roadmap [33], public key infrastructure (PKI) is "the set of hardware, software, people, policies and procedures needed to create, manage, store, distribute, and revoke PKCs based on public-key cryptography". The objective of PKI is to facilitate secure web communications by assuring a correct and proper binding between the identities and their corresponding cryptographic information *i.e.,* public keys. PKI establishes this binding through digitally signed documents called *certificates* that are issued by *certificate authorities (CAs)*.

2.2.1 Public Key Certificates

Public key certificate is a digitally signed document which validates the ownership of a public key. It contains information about the certificate holder *i.e.,* her name and her public key. It also contains the digital signature of a certificate authority (CA) that issues the certificate. CAs are the main components of the PKI scheme that are meant to be *authorities*: that is, they are authoritative over the namespace they bind keys to. While visiting a website that holds a certificate, one can easily verify the signature using the corresponding CA's public key. Successful verification of the signature proves that certificate holder is the real owner of the identity (*i.e.,* `example.com`) and the public key. This leads to establishment of a secure channel between client and server.

2.3 TLS and HTTPS

The Transport Layer Security (TLS) is one of the leading cyprtographic protocols which is widely to secure communications *i.e.,* voice over-IP, email, virtual private networks (VPN) *etc.* over the Internet [34]. In a TLS enabled communication, peers make contact and negotiate their highest cipher suite (ciphersuite negotiation). Then the server authenticate itself to the client (one-way authentication), the authentication method is selected based on the negotiated cipher suite. Eventually, they exchange cryptographic keys which they further use to encrypt communications. Thus, TLS prevent communications from being tampered, forged, and intercepted. The HTTPS is an HTTP protocol which uses SSL/TLS to provide confidentiality, message integrity, and server authentication within for web communications [35]. Originally, HTTPS was only used to secure payment transactions on the web, however, today it is widely used to secure all types of web communications.

2.4 Blockchain

Blockchain technology is an incorruptible digital database which was first introduced as an underlying technology of cryptocurrency Bitcoin in Satoshi Nakamoto's (pseudonym) whitepaper in 2008 [36]. Bitcoin, an electronic payment system, is launched in order to solve the problem of centralization in current payment systems *i.e.,* banks, financial institutions *etc.*— where a central authority is the only authoritative party who is in charge of processing the electronic payments. Placing too much trust on these third parties (TTP) introduces single point of failure — if trusted third party becomes the target of the attackers abuse or deliberately acts maliciously the whole system falls apart. Being a tamper-proof ledger, blockchain maintains the transactions that are entered in a specific network. The ledger is possessed by each member of any specific peer-to-peer network and no centralized version of the information exists. every time a participant (node) creates a transaction on a blockchain it should be first verified by all other nodes in the network using a consensus algorithm such as Bitcoins *proof-of-work (POW).* Bitcoins proof-of-work-based consensus mechanism is used to attain the desired fault tolerance in a decentralized network [37, 38]. In this process, known as *mining*, a group of high computationally power network nodes, known as *miners*, try solve a difficult mathematical puzzle, first node who solves the puzzle is then able to add its proposed block to the blockchain and receive the mining reward [39].

2.4.1 Ethereum

The blockchain technology has primitively gained a wide deployment in area of transactions of digital currencies *e.g.,* Bitcoin cryptocurrency. However, in 2014, Vitalik Buterin represented a new blockchain based application known as Ethereum in his article "Ethereum: A Next-Generation Cryptocurrency and Decentralized Application Platform" [40]. As a blockchain-based distributed public network, Ethereum implements a decentralized virtual machine, known as Ethereum Virtual Machine (EVM), which allows network nodes to execute and deploy programmable smart contracts to the Ethereum blockchain [41]. This new platform enables developers to create and execute blockchain applications called *decentralized applications (dapps)* in a more efficient way.

Decentralized applications are completely open-source and their data is stored in a decentralized manner on the blockchain network. Dapps are created by smart contracts, self-executing contracts that are written in a high level programming language called *Solidity* which is similar to C and JavaScript [42]. Digital smart contracts were first described by Nick Szabo in 1993 [43], however, it reached a high level of adoption by blockchain technology.

2.5 Conclusion

So far this chapter has focused on some of the basic concepts in cryptography. We then gave a brief overview of the blockchain technology and the Ethereum blockchain. In the next chapter, we will represent our thorough empirical study of web certificate model. We will then illustrate and discuss the results that were found during our investigations.

Chapter 3

Understanding the current paradigm for certificate issuance

3.1 Introductory Remarks

HTTPS plays an important role in the maintenance of user privacy when communications take place on the web. Using HTTPS, Internet users can communicate with a web service in a privacy-preserving manner — *i.e.,* the communication channel is private from any other entity that may be privy to the communication channel (*e.g.,* ISPs, mobile carriers, back-bone servers, company/organization gateways *etc.*). However, the compelling guarantees provided by HTTPS rely on a trust model that includes certificate authorities (CAs). Excessive trust that is being placed on these authorities has led them to act as single point of failures — a single malicious or compromised CA can enable HTTPS connections to be vulnerable to eavesdropping, message modification, or injection of malicious scripts. The following issues illustrate the root of the PKI problems on the web.

The number of CAs has exploded. In order to accomplish the trust in a CA based PKI, software companies *e.g.,* Microsoft, Mozilla, Apple, and Opera place a default whitelist of self-signed CA certificates in the firmware of an embedded system and/or browsers as trusted root certificates. Therefore, while visiting an HTTPS website, the browsers merely accept those sites certificates whose validity has been attested by at least one of the trusted root certificates in that whitelist. For instance, Microsoft Windows includes ∼350 trust anchors from ∼140 companies. Trusted root CAs can also issue certificates that authorize other organizations to act as a CA. Thus, the actual number of trusted CAs is greater. Later in this chapter, we show that about 223 companies are browser-accepted.

CAs are targets for privacy breaches. Each trusted CA can issue a browser-acceptable certificate for any site. Hence, an adversary can deceive a weak CA to obtain a fraudulent certificate for a domain he does not own (*i.e.,* `adversary.com`). By doing so, he is able to actively intercept every single communication that takes place between the server hosting `adversary.com` domain and other entities on the web. For example, In 2011, malicious parties were successfully able to illegitimately obtain certificates from the two significant CAs — Comodo [8][9] and DigiNotar [10].

There is also increasing concern that some CAs are vulnerable to certificate compulsion attacks. In this type of attacks, governmental entities force CAs to help them with surveillance by issuing fraudulent certificates for specific website to be spoofed, which they can then use to intercept and tap individuals' secure HTTPS connections to those websites [44]. Surveillance can occur in private networks as well *e.g.,* in enterprises where computers are operated by employers. In this case, organizations can install a root CA certificate on employees' machines which allows them to perform MITM attacks and intercept any HTTPS communication that is established from those computers. Some client-side software, such as anti-virus and parental control, also use a MITM attack to inspect HTTPS content and this can leave the end user vulnerable depending on subtle configuration issues [45].

We know very little about CAs. Although some research has been carried out on HTTPS ecosystem, our knowledge in the full spectrum of CAs is still very limited. What is not yet clear is the number of CAs that actually issue certificates to the public. Additionally, certification practice statements (CPSs), the only techniques that are used by CAs to validate the ownership of domains during the issuance of certificates, are reported to be poor. These techniques have been developed and established by CAs based on their practical experience over a few decades and there is no consensus about the mechanisms that are used by every CA for domain ownership validation.

The purpose of this chapter is to assess and examine real-world CAs on their domain validation practices while issuing certificates. Our first objective is to develop a full enumeration of all uniquely identified trusted CAs that can be traced to a real world entity and issue certificates to external websites referred as *companies* in our work. As a second goal, we attempt to formulate a detailed documentation of the existing validation mechanisms undertaken by companies. The scope of our documentation includes the domain validation techniques performed by companies [46], potential vulnerabilities (*e.g.,* if documented parsing errors can be exploited [47]), personal information that companies collect during a domain validation process, and the costs related to issuing certificates to the public.

The Invisibility of Intermediate CAs. There are two categories of certificate authorities: (i) explicitly trusted root certificate authorities that are recognized by root certificates. These self-signed certificates play a significant role in establishing trust in PKI, they are transmitted to end users by secure physical distribution *i.e.,* being preinstalled on devices. (ii) Implicitly trusted intermediate or subsidiary certificate authorities that have been delegated with the certificate issuing power from root or already-authorized intermediate CAs. There is no solid record of intermediate certificates that currently exist on the web. This is because intermediate certificates can be issued by a root certificate authority (delegating certificate issuing power to another CAs by signing their certificates and creating a certificate 'chain') and this authorization is unknown by anyone until a certificate chain is observed in the wild that uses the intermediate CA. As a result, there is no way to establish how many companies have been issued an intermediate CA. The best we can do within today's PKI is observe as many certificate chains as possible—*e.g.,* by scanning the entire internet or logging certificate chains from consenting users as they browse the web.

3.2 Establishing the Number of Authoritative Entities

To evaluate certificate authorities (CAs) and domain ownership validation procedures they employ, we first need to identify these third parties. These authorities consist of root certificate authorities and intermediate (or subsidiary) certificate authorities. We use other research projects that have measured the internet to enumerate the current list of intermediate certificates that have been seen in the wild. Our analysis begins with collecting a complete record of (i) root and (ii) intermediate certificate authorities:

Root Certificate Authorities. As explained earlier, software vendors such as Microsoft, Mozilla, Apple, and Opera configure extensive lists of built-in root certificates in operating systems and/or browsers, by doing so, they securely transmit the root certificates to the end users. In order to collect the root certificate authorities, between April and June 2015, we retrieved the list of root certificates from (i) Microsoft Windows, (ii) Apple OS X (including OS X Yosemite, iOS 7, iOS 8, Watch OS), and (iii) Mozilla Firefox. After merging these data and collapsing multiple identities for the same CAs, we developed a shortlist of ∼259 actual unique root certificate authorities called *Condensed Root CAs* which we believe is the most comprehensive record of all the current root CAs.

Intermediate Certificate Authorities. Data related to the intermediate certificate authorities on the entire web was collected using the major research projects that have been conducted in the CA ecosystem. (i) We used the data provided by the SSL observatory, an EFF project that investigates the SSL certificates on the Internet [8]. (ii) We employed the data supplied by three projects [48][49][50] that use the ZMap [51], an Internet scanner tool to widely scan and analyze the HTTPS environment. (ii) Lastly, we used known certificate logs from Google's certificate transparency, which provides a monitoring and auditing framework for SSL certificates by appending valid certificates to these logs [52]. The data collected from these project was then parsed and processed and eventually merged into the list called *Intermediate CAs* which contains a list of ∼446 actual unique intermediate certificate authorities.

3.2.1 Analysis and Results

There has been little quantitative analysis of how many of the current certificate authorities (including root and intermediate) are actually unique organizations that provide certificate issuing services to the public. Thus, to gain insight into the domain validation procedures, we first require to identify those CAs who directly provide certificates to the end users, considering the fact that some CAs only issue certificates internally or are managed by a different CA. To do so, we manually research the ~700 intermediate and root certificate authorities that were gathered in the data collection procedures. By visiting these CAs websites, it was found that merely ~223 of these *companies* actually issue certificate to users. Having the list of the companies developed, the final section of this analysis undertakes domain control validation (DCV) methods, deployed by each of these parties. To achieve this for each company, we chose to obtain a regular one year single-domain certificate for a domain name we control. Clearly, we only targeted domain validated (DV) certificates because we did not possess a registered organization nor were we able to be verified in person (to obtain organization validated (OV) and extended validated (EV) certificates). Using the DV certificate purchases as a pilot, we developed a comprehensive enumeration of the companies such as (i) the number of companies that actually issue SSL certificates (EV,OV,DV) to a member of the public, (ii) the cost of acquiring certificates, (iii) the country where each company is located, (iv) contact information of companies, (v) information to be provided to CAs during the domain ownership verification, (vi) how this information is used and (viii) how is identity of users validated.

In the first stage of our scan, we contact each company in the list to obtain a certificate for our domain, we discover that only 71 of these companies actually issue at least one of the EV, OV, DV certificates to the public users. The other 152 companies cannot provide certificates to us due to the following reasons:

- They only issue certificates to their own country services *e.g.,* universities and research institutes.
- They only issue certificates to services from the national government agencies.
- They only issue certificates in projects.
- They only issue certificates to the external partners/vendors that do business

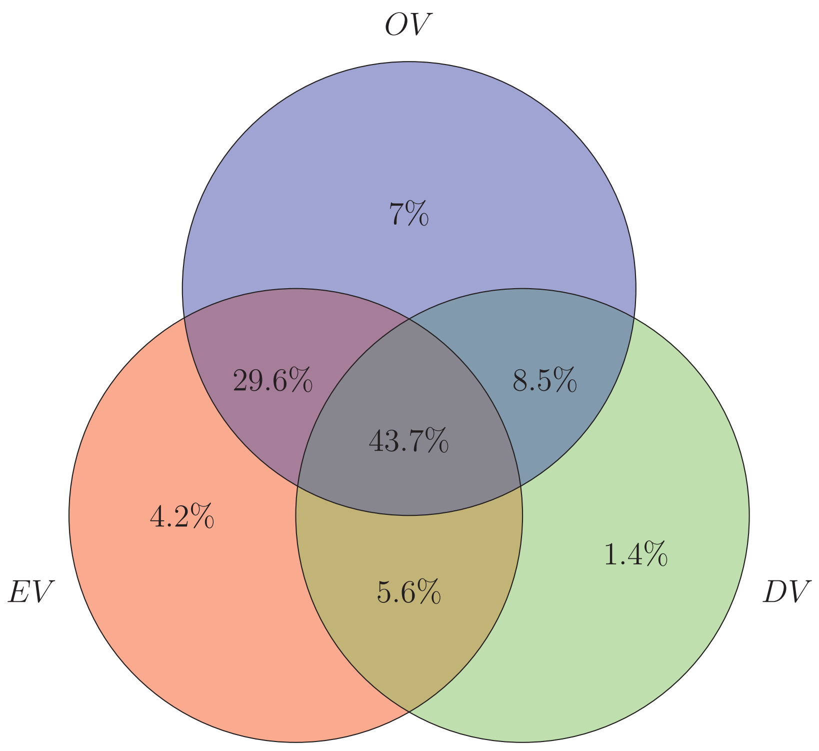


Figure 3.1: Distribution of the certificates issued by 71 companies — We found that a large group of these companies only issue the high assurance (OV and EV) certificates.

with their company.

- They require a face-to-face registration for any type of web server certificates. This is either at their office,or via a notary in applicants country.

Therefore, we complete our survey by contacting 71 companies that assuredly provide certification services to us (as a public user from a foreign country). The results of our survey indicate that among 71 companies, 42 of them are willing to issue domain validated (DV) certificates to end users (see Figure 3.1).

Furthermore, results reveal that 73.3% of these companies issue extended validated (EV) and organization validated (OV) certificates, which are both considered as high assurance certificates. The reason behind this is, as explained earlier, due to the existing drawbacks in domain control validation (DCV) procedures that allow malicious parties to obtain fraudulent certificates for domains they do not control. By acquiring a fake certificate for victims domains, adversaries can perform MITM attacks and intercept traffics before victims can receive it. This is while DV certificates are often preferred above all other certificates because (i) they are offered at much lower prices than high assurance (EV and OV) certificates and (ii) they are issued during effortless and automated processes. We also found that 33.3% of the companies that issue DV certificates are currently located in the United States.

After reviewing each company's certificate practice statements (CPS), we discovered the companies do not always document their certificate issuance policies in a precise manner. In some cases, verification procedures that companies provide in their CPSs are not equivalent to what they offer in practice.

3.3 Showcasing the Indirection Used by CAs

Domain control validation (DCV) procedures are a set of techniques employed by certificate authorities to verify the ownership of a (sub)domain that certificate is asked for. Unlike high assurance certificates, domain validated (DV) certificates only represent a certificate holder's control over a given domain name and they do not provide any intuition about the identity of the owner. DCV can be performed using any one of the (i) email-based, (ii) HTTP-based, and (iii) DNS-based methods. What follows is a brief description of the DCV methods and the security issues with them.

Email-based Validation. Email-based validation is the most supported method of host name verification. In this approach, CA sends a unique nonce as a secret challenge to an email address it is assumed to belong to the subscriber. If the certificate applicant can access this challenge, it is considered to have proven ownership of the domain and causes the CA to issue the certificate. CAs specify a list of acceptable addresses for a given domain using a few generic names including `admin`, `administrator`, `hostmaster`, `postmaster`, and `webmaster`, in addition to any email addresses that appears on a domain's WHOIS record.

HTTP-based Validation. In this technique, the CA delivers a unique, non-secret nonce (or hash of the certificate request) to the subscriber through an HTTPS channel. In order to prove domain name ownership, the subscriber needs to create a text file with the nonce and upload the file to the root directory of his domain that is to be validated. In the final part of the verification process, CA checks the presence of the text file. If the file is successfully retrieved by the CA and its contents match, domain control is confirmed.

DNS-based Validation. Using the DNS based method, the subscriber receives a unique, non-secret nonce (or hash of the certificate request) from CA through an HTTPS channel and he is required to publicize the nonce in his DNS CNAME record for the domain. Afterward, the CA queries the corresponding name servers to verify the presence of the DNS CNAME record. If the record is successfully obtained, domain ownership is validated.

3.3.1 Authoritative Issue

In the current web certificate model, certificate authorities are meant to be *authorities*: that is, they are authoritative over the namespace they bind keys to. The reality is that the web still runs largely on domain validated certificates [53][54] and for domain validation, certificate authorities generally are not any more or less authoritative over who owns what domain than you or I. There is arguably no single party that is authoritative. ICANN manages the top-level domains and delegate registration of domains to registrars. Registrars sell domains to companies or individuals. Registrars do not

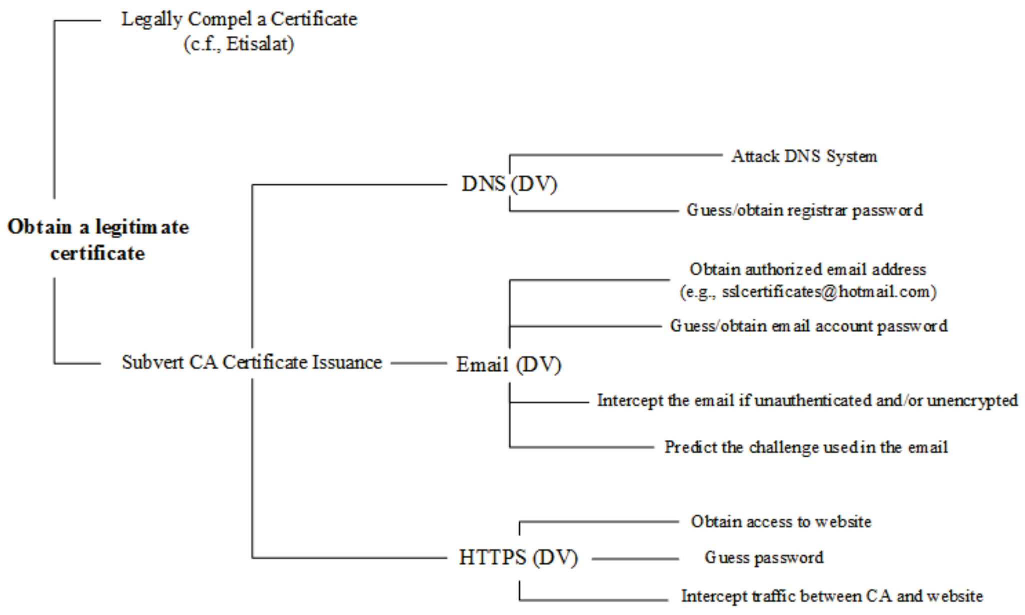

Legally Compel a Certificate (c.f., Etisalat)
Obtain a legitimate certificate
Subvert CA Certificate Issuance
DNS (DV)
Attack DNS System
Guess/obtain registrar password
Email (DV)
Obtain authorized email address (e.g., sslcertificates@hotmail.com)
Guess/obtain email account password
Intercept the email if unauthenticated and/or unencrypted
Predict the challenge used in the email
HTTPS (DV)
Obtain access to website
Guess password
Intercept traffic between CA and website

Figure 3.2: Possible attacks on the DCV techniques.

identify the people buying the domains; instead they have the purchaser set a username/password for an account that they can use to manage the domain. Registrars are natural entities to serve the role of a CA and indeed there is overlap between the set of registrars and the set of CAs, however a registrar/CA is not restricted to issuing certificates to only its own customers and can in fact issue certificates for any domain. Many CAs are not registrars or connected in any way to the domain management eco-system. They establish who owns a domain through indirection.

3.3.2 Indirection Issue

A central problem with the certificate model is that CAs resort to *indirection* to issue certificates because they are not directly authoritative over who owns what domain. For example, email based DCV involves 2 levels of indirection: (1) CAs appeal to DNS to establish the MX record of the domain (*i.e.,* the subscriber's mail server's IP address); and (2) CAs appeal to SMTP to establish a communication channel to the subscriber. For every level of indirection, there are a set of vulnerabilities which might allow a malicious party to break the verification process and obtain a fraudulent certificate for a domain they do not own.

To illustrate this in more detail, we will consider email-based validation as described above. We note that email validation works in conjunction with DNS, so the vulnerabilities on this process subsume the set of vulnerabilities of DNS-based validation. We leave aside HTTP-based validation as it is largely similar; but we note one interesting issue: the CA must fetch the posted hash of the certificate request from the website and since the website is trying to obtain a certificate, it follows that they likely do not have an existing certificate and therefore are not providing this file over HTTPS. Any man-in-the-middle between the webserver and the CA could respond to the CA's request with a fraudulent CSR (even if it doesn't actually exist on the webserver) and obtain a fraudulent certificate.

We now enumerate at list of threats to email-based validation to illustrate how broad the attack surface becomes when excessive indirection is relied on. These are summarized in Figure 3.2.

1. **Reserved Emails:** A CA specifies a list of email addresses to receive the challenge. The underlying assumption is that only the domain owner controls this address. However the domain owner might not reserve that email address or even be aware that a certain email address is being used by one of the CAs for this purpose. And recall that just a single CA needs to use a single non-standard email address (*e.g.,* a translation of administrator into their local language) to open up this vulnerability. For example, Microsoft's public webmail service `login.live.com` saw an attacker successfully validate his ownership of the domain using an email address `sslcertificates@live.com` which was open to public registration [55].

2. **Whois Emails:** A CA will also optionally draw the email address from the Whois record for the domain. A domain's whois record is generally protected by the username/password set by the domain owner with their registrar. Any attack on this password (*e.g.,* guessing or resetting) or directly on the account (*e.g.,* social engineering [56]) would allow the adversary to specify an email address that they control.

3. **MX Record:** A CA will establish the IP address of the mailserver from the MX record for the domain. As above, all domain records including the MX record is managed through the owner's account with her domain registrar. Any method for obtaining unauthorized access to this account would enable an adversary to list their own server in the MX record and receive the email from the CA.

4. **DNS Records:** If an adversary cannot directly change a DNS record, they might conduct other attacks on the CA's view of DNS. For example, they might employ DNS cache poisoning which can result in invalid DNS resolution [57]. They might also exploit an available dangling DNS record (Dare) [58]. A Dare occurs when data in a DNS record (such as CNAME, A, or MX) becomes invalid but is not removed by the domain owner. For example, if the domain owner forgets to remove the MX record (the IP address of the server) from DNS, the associated DNS MX record is said to be dangling. If an adversary can acquire this IP address at some future point, he is able to redirect all traffic intended for the original domain to his server, including information sufficient for a CA's domain validation process. Thus a malicious party can use a Dare to obtain a fraudulent certificate.

5. **SMTP:** Once the CA establishes the mailserver's record, it will send the email to the mailserver with SMTP (the standard protocol for transfer of email).

Since the email contains a secret nonce, confidentiality of this email is crucial. SMTP uses opportunistic encryption that is not secure against an active adversary. Thus a man-in-the-middle between the CA's mailserver and the ultimate destination (including an forwarding mailservers) could request a fraudulent certificate, intercept the ensuing email, reply with the correct nonce, and be issued the fraudulent certificate.

6. **Email Accounts:** Email accounts are generally protected with a username and password (over IMAP or POP3) to prevent unauthorized access. In some cases, they might be protected with a client certificate. An adversary who can gain access to any one of the accounts that should be reserved by the domain owner (*e.g.,* texttadmin, `hostmaster`, `webmaster`, *etc.*) could obtain a fraudulent certificate for that site. This could include guessing or resetting the password, using social engineering, or obtaining access to the server hosting the email for the account.

3.4 Conclusion

In this chapter, we developed a complete enumeration of all the existing certificate authorities. We then surveyed the procedures these CAs enforce in order to validate domain name ownership during the certificate issuing processes. It has been discovered that while domain validation methods seem promising, they are not completely secure in practice and can be compromised by malicious parties on the web. As a result, attackers are able to deceive and/or compromise the CAs to sign invalid certificates on behalf of them. We also discussed the major issue with the the current certification system is that the entities that issue a certificates for domain names (CAs) are not the actual domain's owners. We believe that certificate authorities are not special and any entity that owns a domain can be a CA and issue a certificate for that domain. In fact, Verisign is not more authoritative than anyone else in the case of validating domain names.

As discussed above, these third parties (CAs) are vulnerable to a vast group of vulnerabilities that if compromised, an adversary obtains a certificate for a domain that does not belong to him. This issue can be addressed by designing a system in which the domain owner can issue and manage certificates without having to rely on any third party. By removing the CAs from this infrastructure and make entities authoritative over their domain names and certificates, we can eliminate single point of failures and their prominent consequences from the public key infrastructure.

In the next chapter, we introduce our system which is a new paradigm for the certification model. Using this new system, we will be able to eliminate certain limitations of the current CA model. As it was discussed in Section 3.1, currently a large number of CAs ($\sim$300) exist on the web and this is while the root CAs can still delegate their issuing certificate power to the other CAs. However, our proposed scheme eliminates the 300 CAs and replaces it with *one* decentralized, authoritative system. Further, the system is fully authoritative over domain names and cryptographic keys and so no indirection is necessary.

Chapter 4

A New Uni-Authoritative Paradigm for Certificate Issuance

4.1 Introductory Remarks

In the previous chapter we discussed domain control verification (DCV) techniques and the primary issues with these procedures. Our findings were largely based upon our empirical study that investigates how the certificate authorities actually issue certificates to the end users and how they verify the ownership of domain names. Actual failures in domain validation procedures occur when CAs are not effectively successful in verifying the subjects identities, therefore, enabling the registration of a public key under another entity's already-registered domain name.

Issuing certificates is equal to binding names from a namespace to cryptographic keys. To provide this binding, a system can either rely on indirection and have certificate authorities that are not authoritative over the namespace try to verify ownership; or it can collapse the indirection if the issuing entity is authoritative over the namespace—we call this the uni-authoritative paradigm (see Figure 4.1). In the cur-

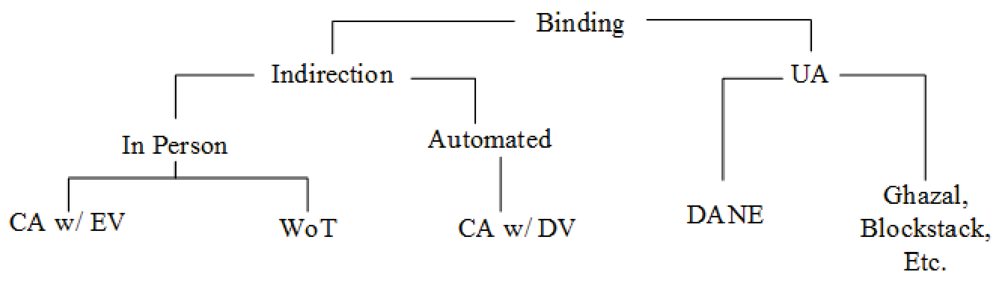


Figure 4.1: Different approaches to provide the binding between a namespace and cryptographic public keys.

rent web certification model, CAs heavily rely on indirection to verify the ownership of the namespace and provide the binding. Technically, they can validate the namespace ownership via (i) DCV methods which are typically automated and effortless and/or (ii) in-person validation techniques. In-person includes high assurance certificates (EV,OV) which are validated by the CA using government-issued documents, or Web-of-Trust (WoT) where external parties validate and sign off on one's identity typically also after checking some government-issued ID. Note that in-person validation is still indirection as the government is authoritative over its citizen's names or registered business names, while CAs are typically non-government entities (and even when CAs are governmental, they are authoritative over all domains including ones not owned by their citizens). From here forward, we consider the case of automated domain validated certificates.

In the uni-authoritative paradigm, the owner of the namespace is fully authoritative over it and is the same entity that binds names to cryptographic keys. In this thesis, we explore this in the context of blockchain technology. If a PKI were added to a blockchain, who would be authoritative over the namespace of domain names? When domain names themselves are issued through the blockchain (*e.g.,* Namecoin), then the blockchain is actually the authoritative entity. Such a design would thus be uni-authoritative.

Arguably, indirection can be collapsed in the traditional web certificate model as well. As we argued in the previous chapter, DNS (in conjunction with ICANN) is authoritative over the namespace of domain names. If ICANN/DNS held key bindings, there would be no indirection or CAs needed. Indeed exactly this has been proposed under the same of DNS-based Authentication of Named Entities (DANE). Thus blockchains and DANE are both examples of a uni-authoritative paradigm. A deployment issue with DANE is that DNS records do not generally have message integrity (except via the under-deployed DNSSEC) whereas blockchain transactions do.

As it was mentioned in the previous chapter, a variety of attacks in the current CA ecosystem show these third parties' inability to provide an authentic and proper binding between the namespaces and the cryptographic keys — where adversaries were able to acquire fraudulent certificates for domains they do not control. incorrect bindings can be either (i) prevented or (ii) detected. As it can be seen in Figure 4.2, prevention techniques can be applied by any CAs and/or any sites to prevent from a wrong binding. For example, a domain owner could add a CAA record to his DNS where he declares a list of CAs that are allowed to issue a certificate for his domain (IKP is a blockchain analogue); he could hardcode his certificate or constraints on his certificate into the browser the user will install (*e.g.,* key pins in Google Chrome); or he could pin his certificate the first time the user visits his site to protect subsequent visits (TACK) assuming the first interaction is trustworthy (trust-on-first-use or TOFU).

Unlike prevention techniques, detection methods do not prevent wrong bindings from occurring, instead they add visibility and transparency about (namespace, keys) binding, so it can be detected in case of failures. In the traditional web certification model, CAs do not provide any insight about this binding. Lack of the visibility of

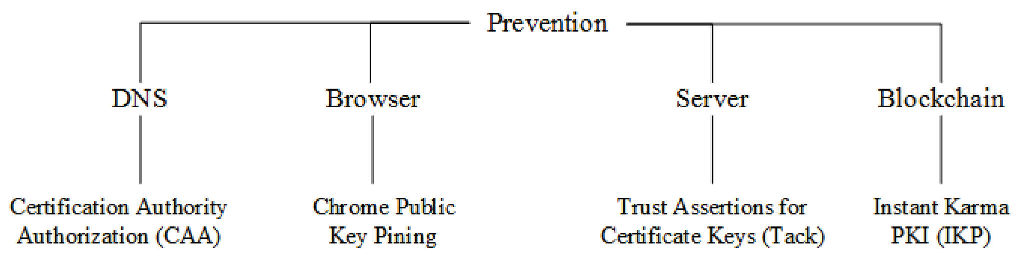

Prevention
DNS
Browser
Server
Blockchain
Certification Authority Authorization (CAA)
Chrome Public Key Pining
Trust Assertions for Certificate Keys (Tack)
Instant Karma PKI (IKP)

Figure 4.2: Techniques to prevent from incorrect (namespace, keys) bindings.

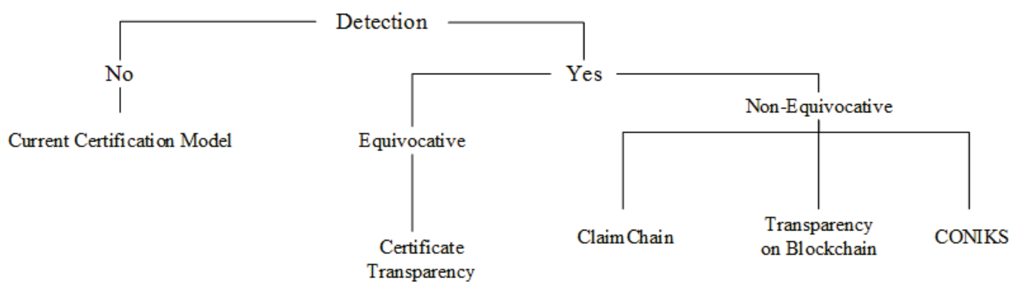


Figure 4.3: Techniques to add visibility about (namespace, keys) bindings.

bindings has led to several failures; thus, attempts have been made to add transparency to the CAs ecosystem (see Figure 4.3). Certificate transparency (CT) [19], sovereign keys (SKs) [20], and ARPKI [21] are systems that augment the current web certificate mode by supplying a log of CA-issued certificates. A step further is not to rely on CAs at all; another group of transparency solutions rely on users validating their own entry in the log and then enforcing that all users see the same entry when referencing the log even if the server hosting the log is malicious (a property called non-equivocation). CONIKS is such a system and provides a distributed log but for public keys [23]. ClaimChain is similar to CONIKS but finds a middle-ground between using a small set of distributed servers (CONIKS) and a fully decentralized but global state (blockchain) by having fully decentralized, local states that can be cross-validated [25].

4.2 The **Ghazal*** System

Our proposed scheme is entitled Ghazal*, a smart contract-based naming and PKI uni-authoritative system. [1] Ghazal* is is actually a new uni-authoritative paradigm that resolves some of the fundamental issues with the current certification model — authority and indirection. It enables entities, whether they are people or organizations, to fully manage and maintain control of their domain name without relying on trusted third parties. By proposing Ghazal*, we argue that adding programmability to a dapp-based PKI provides benefits beyond using the blockchain as an append-only broadcast channel.

Using our system, users can register unclaimed domain names as globally readable identifiers on the Ethereum blockchain, bind the domain name to arbitrary data *i.e.,* public keys *etc.* These values are globally readable, non-equivocating, and not vulnerable to the indirection attacks outlined in chapter 3. Anyone can claim a domain on a first-come, first-serve basis. Because it is decentralized, names cannot be re-assigned without the cooperation of the owner (whereas an ICANN address like `davidduchovny.com` can be re-assigned through adminstrative mediation). Another feature of Ghazal* system is speeding up the DNS updates. In our system, DNS resource records are updated in 12 seconds (block interval in the Ethereum blockchain), whereas it would take 3 days for users to update domain names resource records using the traditional DNS system.

This novel system consists of two essential elements. First, the smart contract that resides on the Ethereum blockchain and serves as the interface between entities and the underlying blockchain. The second primary component of the system are the clients, including people or organizations that interact with Ghazal* smart contract in order to manage their domain names. Figure 4.4 represents the primary states a

[1] `https://github.com/mahsamoosavi/Ghazal`

domain name can be in and how state transitions work. These states are enforced within the code itself to help mitigate software security issues related to unintended execution paths.

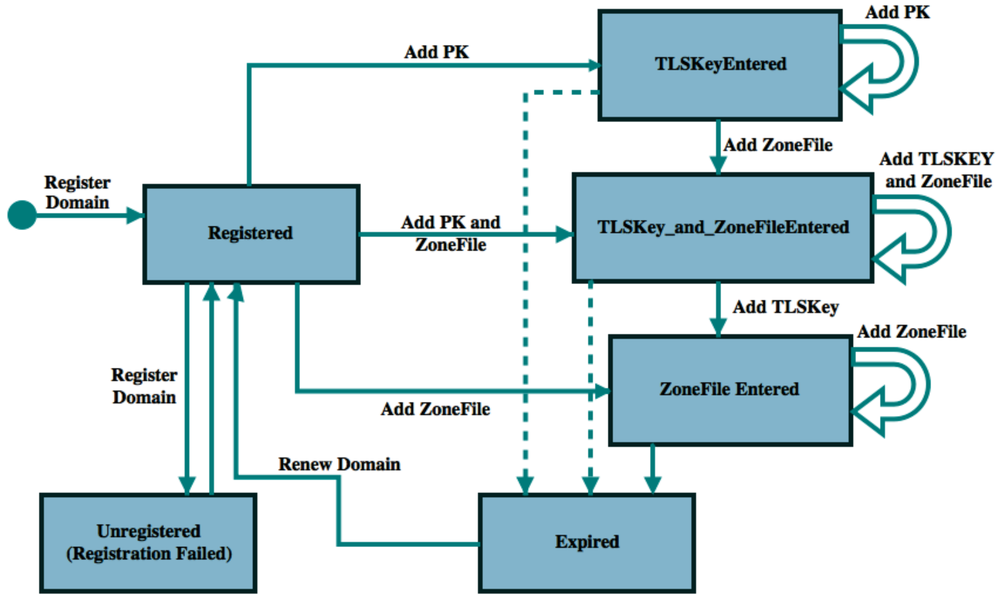


Figure 4.4: Primary states and transitions for a domain name in Ghazal*.

4.2.1 Exploring **Ghazal*** design choices

Beyond simply presenting our design, we think it is useful to explore the landscape of possible designs. To this end, we discuss some deployment issues that we faced where there was no obvious "one right answer." These are likely to be faced by others working in this space (whether working narrowly on PKI or broad identity on blockchain solutions).

4.2.1.1 Design Decision #1: Domain Name Expiration

Typically domain name ownership eventually *expires*. Once a domain expires, it is returned to the primary market, except if the users renews it. However, expiration does not necessarily have to mean a disclaimer of ownership; there are other options.

1. **Domain names never expire and last forever.** Designing a system with no domain name expiration would be highly vulnerable to domain squatting. Domain squatting is registering domain names in speculation that the will increase in value. These domain names generally do not point to any relevant IP address (except to earn revenue on accidental visits). If domain names never expire, squatting may be significantly problematic as squatted names would be locked forever while legitimate users will end up choosing unusual names from the remaining namespace. To be clear, even without expiration, if domains are cheap, squatting is problematic (*e.g.,* Namecoin [15]).

2. **Domain names get deleted once they expire, except being renewed by the user.** The most restrictive system design is where a domain name effectively gets deleted and is returned to the registry of unclaimed names once it expires, unless the user renews it. This model has the following two issues. First, if a browser tries to resolve an expired domain, because the blockchain has a complete, immutable history of that domain, we would expect users to want it resolved according to the previous owner. Rolling back expiration is possible in a way not supported by DNS and it resolves simple human errors of forgetting to renew domains, so we do not expect browsers to necessarily fail when it could make a sensible guess as to which server their users are looking for. The second reason to drop the deletion model of expiration is that Ethereum contracts can only run when a function is called. If no one calls a function at expiration time, the contract cannot self-execute to modify itself. The fact that it is expired can be inferred from contract if it includes a time but the contract itself will not transition states until someone calls a function that touches that particular contract. An alternative is to rely on a third party like Ethereum Alarm Clock [59] for scheduling future function calls. This is suitable only if

the threat model permits relying on a trusted third party and a single point of failure (for this one feature).

3. **Control over domain names is lost once they expire, except being renewed by user.** In Ghazal*, expired domains continue to function although the owner (i) looses the sole claim to that domain and cannot preserve it if someone else purchases it, and (ii) she cannot modify the domain in anyway (*e.g.,* add certificates or change zone information) unless if she first renews it. Essentially, purchasing a domain name does not entitle an entity to own it forever; expired domain names are returned back to the primary market and are available for all the users within the system. However since a full history of a domain is present, the system's best effort at resolving the domain will be to preserve the last known state. Expiration in conjunction to the amount of the fee will influence the degree of domain squatting, and having expiration at all will allow abandoned domains to churn if they are under demand.

4.2.1.2 Design Decision #2: Registration Fees

In Ghazal*, new registrations and renewals require a fee. This fee is a deterrent against domain squatting. The fee amount is difficult to set and no fee will be perfectly priced to be exactly too high for squatters but low enough for all 'legitimate' users. Rather it will trade-off the number of squatters with the number of would-be legitimate users who cannot pay the fee. Namecoin is evidently too cheap and ICANN rates seem reasonable. We leave this as a free parameter of the system. The important decisions are: (1) in what currency are they paid and (2) to whom. Every Ethereum-based system, even without a fee, will at least require gas costs. Additional fees could be paid in Ether or in some system-specific token. Since it is a decentralized system and the fee is not subsidizing the efforts of any entity involved, there is no one in particular to pay. The fee could be paid to an arbitrary entity (the system designer or a charity), burned (made unrecoverable), or to the miners. In Ghazal*, fees are paid in Ether and are released to the miner that includes the transaction in the blockchain.

4.2.1.3 Design Decision #3: Domain Name Renewal

We design Ghazal* in such a way that the domain owners can renew their domains before their validity period comes to an end, however they cannot renew an arbitrary number of times. Specifically, a renewal period becomes active after the domain is past 3/4 of its validity period. Renewal pushes the expiration time forward by one addition of the validity period (thus renewing at the start or end of the renewal period is inconsequential and results in the domain having the same expiration time). Requiring renewal keeps users returning regularly to maintain domains, and unused domains naturally churn within the system. Domain name redemption period can take different values. We experiment with a validity period of 1 year; thus, the renewal period would start after 9 months and last 3 months.

4.2.1.4 Design Decision #4: Domain Name Ownership Transfer

In Ghazal*, domain owners can transfer the ownership of their unexpired domains to new entities within the system. Basically, transferring a domain name at the Ethereum level means changing the address of the Ethereum account that controls the domain. Our system offers two ways of transferring the ownership of a domain:

1. **Auctioning off the domain name.** A domain owner can voluntarily auction off an unexpired domain. Once an auction is over, the domain is transferred to the highest bidder, the payment goes to the previous owner of the domain, and the validity period is unaffected by the transfer (to prevent people from shortcutting renewal fees by selling to themselves for less than the fee). If there

```
//Possible states of every auction.
enum Stages {Opened, Locked, Ended}

    struct AuctionStruct
    {  uint CreationTime;
        address Owner;
        uint highestBid;
        address highestBidder;
        address Winner;
        Stages stage;
        //To return the bids that were overbid.
        mapping(address => uint) pendingReturns;
        //To return the deposits the bidders made.
        mapping(address => uint) deposits;
        //Once an address bids in the auction, its associated boolean value will be
            set to true within the "already_bid" mapping.
        mapping(address => bool) already_bid;
        bool AuctionisValue;
      }
//AuctionLists mappings store AuctionStructs.
mapping (bytes32 => AuctionStruct) internal AuctionLists;
```

Code 4.1: Implementation of AuctionStruct and AuctionLists mapping in Ghazal* smart contract.

are no bidders or if the bids do not reach a reserve value, the domain is returned to the original owner. While under auction, a domain can be modified as normal but transfers and auctions are not permitted. To implement the auction feature, we use the fact that Solidity is object-oriented. We first deploy a basic Ghazal function without advanced features like auctions, and then use *inheritance* to create a child contract Ghazal* that adds the auction process. Using Ghazal*, a user can run any number of auctions on any number of domains he owns. This is implemented through a mapping data structure called *AuctionLists* to store every auctions along with its attributes. *AuctionLists* accepts *Domain names* as its keys, and the *AuctionStructs* as the values (see Code 4.1). Using the mapping and Ethereum state machine, we enforce rules to prevent malicious behaviors *e.g.,* domain owners can auction off a domain only if there is no other auction running on the same domain. To encourage winners to pay, all bidders must deposit a bounty in Ether the first time they bid in an auction (amount set by the seller). This is refunded to the losers after bidding closes, and to the winner after paying for the domain. Without this, users might disrupt an auction by submitting high bids with no intention of paying.

```
modifier CheckDomainExpiry(bytes32 _DomainName) {
       if (Domains[_DomainName].isValue == false)
          {Domains[_DomainName].state=States.Unregistered;}
       if (now>=Domains[_DomainName].RegistrationTime+10 minutes)
          {Domains[_DomainName].state = States.Expired;}
        _;
    }
modifier Not_AtStage(bytes32 _DomainName, States stage_1, States stage_2) {
        require (Domains[_DomainName].state != stage_1 && Domains[_DomainName].
            state != stage_2);
        _;
    }
modifier OnlyOwner(bytes32 _DomainName) {
        require(Domains[_DomainName].DomainOwner == msg.sender);
        _;
    }
function Transfer_Domain(string _DomainName,address _Reciever,bytes32 _TLSKey,
    string _IP_Adress) public
CheckDomainExpiry(stringToBytes32(_DomainName))
Not_AtStage(stringToBytes32(_DomainName),States.Unregistered,States.Expired)
OnlyOwner(stringToBytes32(_DomainName))
    {
        DomainName = stringToBytes32(_DomainName);
        Domains[DomainName].DomainOwner = _Reciever;
        if (_TLSKey == 0 && stringToBytes32(_IP_Address) != 0) { Wipe_TLSKeys(
            DomainName); }
        if (stringToBytes32(_IP_Address) == 0 && _TLSKey != 0 ) {
            Wipe_IP_address(DomainName); }
        if (stringToBytes32(_IP_Address) == 0 && _TLSKey == 0 ) {
            Wipe_TLSKeys_and_IP_address(DomainName); }
    }
```

Code 4.2: Transfer_Domain function of Ghazal* smart contract.

2. **Transfer the ownership of a domain name.** A domain owner can also transfer an unexpired domain to the new Ethereum account by calling the

Transfer_Domain function which simply changes the Ethereum address that controls the domain name. The owners can also decide to either transfer domain's associated attributes (*e.g.,* TLS certificates) or not, when they transfer the domain. This is possible with either supplying these attributes with zero or other desired values when calling the `Transfer_Domain` function (see Code 4.2).

To prevent from MITM attacks, TLS certificates should be revoked once a domain name is transferred. However, security incidents reveal that this is not commonly enforced in the current PKI. For instance, Facebook acquired the domain `fb.com` for $8.5M in 2010, yet no one can be assured if that the previous owner does not have a valid unexpired certificate bound to this domain [14]. This has been successfully enforced in our system as the new owner of the domain is capable of modifying the domain's associated TLS keys, which results in protecting communications between the clients and his server from eavesdropping.

4.2.1.5 Design Decision #5: Toward Lightweight Certificate Revocation

In the broader PKI literature, there are four traditional approaches to revocation [60]: certificate revocation lists, online certificate status checking, trusted directories, and short-lived certificates. Revocation in the web certificate model is not effective. It was built initially with revocation lists and status checking, but the difficulty of routinely obtaining lists and the frequent unavailability of responders led to browsers failing open when revocation could not be checked. Some browsers build in revocation lists, but are limited in scope; EV certificates have stricter requirements; and some research has suggested deploying short-lived certificates (*e.g.,* four days) that requires the certificate holders to frequently renew them [61] (in this case, certificates are not explicitly revoked, they are just not renewed). Which model does a blockchain implement? At first glance, most blockchain implementations would implement a trusted directory: that is, a public key binding is valid as long as it is present and revocation simply removes it. The issue with this approach on a blockchain is how users establish they have the most recent state. With the most recent state in hand, revocation status can be checked. This check is potentially more efficient than downloading the entire blockchain (this functionality exists for Bitcoin where it is called SPV and is a work in progress for Ethereum where it is called LES). However a malicious LES server can always forward the state immediately preceding a revocation action and the client cannot easily validate it is being deceived.

At a foundational level, most revocation uses a `permit-override` approach where the default state is permissive and an explicit action (revocation) is required. Short-lived certificates (and a closely related approach of stapling a CA-signed certificate status to a certificate) are `deny-override` meaning the default position is to assume a certificate is revoked unless if there is positive proof it is not. This latter approach is better for lightweight blockchain clients as LES servers can always lie through omitting data, but cannot lie by including fraudulent data (without expending considerable computational work). As an alternative or compliment, clients could also take the consensus of several LES servers, although this 'multi-path probing' approach has some performance penalties (it has been suggested within the web certificate model as Perspectives [62] and Convergence [63]).

In Ghazal*, public keys that are added to a domain name expire after a maximum lifetime, *e.g.,* four days. Expiration is not an explicit change of state but is inferred from the most recent renewal time. Owners need to rerun the key binding function every several days to renew this. If an owner wants to revoke a key, she simply fails to renew. To verify the validity of a certificate, one is now able to use a LES-esque protocol. Once a user queries a semi-trusted LES node for a corresponding record of a domain, the node can either return a public key that is four days old, which user will assume is revoked, or a record that newer that the user will assume is not revoked. Although this approach requires the frequent renewal of public keys, it is a cost that scales in the number of domains as opposed to revocation checks which scale in the number of users accesses a domain.

4.3 Ghazal Main Operations

At the time of writing this thesis, there are 19 functions in Ghazal* (Ghazal's child contract that adds the auction process). In this section, we represent a list of primary functions that are mainly used to mange (*e.g.,* register, renew, *etc.*) the domain names within our system.

Domain Name Registration. *Register* function allows entities to register unregistered or expired domain names.This function is *payable* that is, entities need to pay the domain registration fees to call this function and claim the domain names. *Register* function takes one parameter as its input; **Domain name**– A string representing a domain name the user aims to claim.

Domain Name Renewal. *Renew* function allows entities to renew their domain names before their validity period comes to an end.This function is *payable* and takes one parameter as its input; **Domain name**– A string representing a domain name the owner wants to renew.

Add Certificate. *Add TLSKey* function allows domain name owner to bind TLS keys to his domain name. This function can be also used to overwrite the existing TLS keys in case of the private key loss or interception. It takes the followings as input:

Domain name– A string representing a domain name to which the owner aims to bind the TLS keys.

TLS Key– A dynamically sized byte array that stores TLS keys.

Add Zonefile. Using the *Add Zonefile* function, domain owner can add the associated resource records to his domain name. Followings represent the input parameters of this function:

Domain Name. A string representing a domain name to which the owner aims to the resource records.

IP Address. An string representing the domain name's associated IP address.

Add Certificate & Zonefile. *Add TLSKey & Zonefile* function allows domain owners to bind TLS keys and zone files to their domain names simultaneously. This function takes the three following input parameters:

Domain Name. A string representing a domain name to which the owner aims to add the attributes.

IP Address. A string representing the domain name's associated IP address.

TLS Key. A dynamically sized byte array that stores TLS keys.

Revoke Certificates. *Revoke Certificate* function allows domain owners to delete any specific certificate that is bound to their domains. This function takes two input parameters:

Domain Name. A string representing a domain name.

TLS Key. A dynamically sized byte array representing the TLS key that owner wants to revoke.

4.4 Walkthrough of Domain Name Resolution in Ghazal*

This section provides a walkthrough of resolving a domain using the Ghazal* system.

4.4.1 Ghazal* with `.ghazal` namespace

In order to visit `bank.ghazal` website over HTTPS, the user's browser needs to resolve the IP address for `bank.ghazal` and obtain its public key. The browser might be configured in one of two ways to resolve domains—one method is more secure but has deployability challenges, while the other makes a trust assumption and gains efficiency. The first option is for the user's client to maintain a local copy of the Ethereum blockchain. In this case, the browser can query its local copy of the blockchain and parse the mapping data structure to recover the domain name's associated attributes (*e.g.,* DNS resource records and TLS certificates). Due to the data structure, accessing these values is performed in constant time because the mapping works like a hash table (so the entire list of domains does not have to be searched, as would be the case with an array—see Section 5.3.4). Additionally, because the blockchain is local, this introduces no extra rounds of communication (other than having the blockchain updated). This approach is the most secure way of resolving `.ghazal` domain names as the clients directly consult with the blockchain without relying on any other party. However, it requires clients to download and maintain the entire Ethereum blockchain, which is not feasible for lightweight blockchain clients such as smartphones. Another way of resolving the `bank.ghazal` domain name is to rely on semi-trusted full node LES servers. In this approach, instead of downloading the entire Ethereum blockchain, browsers can connect to LES severs and obtain the domains attributes. This check is more efficient than downloading the entire blockchain, however it is less secure as trust is involves trusting other parties who can be malicious. See Section 4.2.1.5 for more on this approach.

4.5 Conclusion

In this chapter, we pinpointed and categorized existing solutions to the CAs ecosystem failures. We then described our system Ghazal*, a naming and PKI uni-authoritative system which is implemented on the Ethereum blockchain. The last sections of this chapter, we thoroughly explained our system's design choices as well as its primary operations.

In the chapter 5 we will further explain the implementation and deployment of Ghazal*.

Chapter 5

Implementation and Deployment of **Ghazal***

5.1 Introductory Remarks

In the previous chapter, we described Ghazal*, a new paradigm for a DNS system and certification model which allows entities to securely register domain names and obtain certificates for those names. We also discussed the various design decisions that have been taken into account while designing Ghazal* scheme. While Bitcoin scripting language is highly restrictive, the Ethereum blockchain is developed to enable developers to leverage the underlying blockchain security and execute their own programs, known as smart contracts, in a fully decentralized manner. In this chapter, we further detail the implementation and deployment of our system, that is mainly a smart contract written in Solidity, in addition to identifying the security requirements of its functions.

5.2 Ethereum Concepts

What follows is a description of the Ethereum blockchian concepts which draw upon insights into how this technology works and what are the chief components of it.

5.2.1 Ethereum Accounts

Accounts are entities that play the most significant role in the Ethereum blockchain. There are two types of (i) externally owned accounts and (ii) contract accounts on the Ethereum. Both types of Ethereum accounts are associated with a 40-character hexadecimal format public key known as Ethereum address. These accounts, as Ethereum network entities, hold states. Externally owned accounts, referred to accounts, own balance while contract accounts hold balance and contract storage. Ethereum nodes keep track of the network's state which is the most recent state of each existing account and is updated with every block that is added to the blockchian.

5.2.2 Smart Contracts and Transactions

As mentioned, a contract is an Ethereum account that contains a piece of code and can be executed on the Ethereum Virtual Machine (EVM). Once deployed on the Ethereum, smart contracts are not executed unless they are called up and triggered by mechanisms known as transactions. Transactions can be originated from either another contracts or normal accounts, in both cases the contract code is executed on the EVM and its state changes based on the transactions it has received as an input.

Originally, Smart contracts were written and developed high level languages including Mutan (C-like language) [64], LLL (LISP-like language) [65], and Serpent (Python-like language) [66]. However, smart contracts are currently written in Solidity, a high level object oriented language which is similar to Java [67].

5.2.3 Ether

Similar to bitcoin, ether (ETH) is a class of cryptocurrency, while it alternatively supplies "fuel" for the Ethereum network to run the decentralized applications. In order for the Ethereum nodes to process and execute a transaction, a transaction fee is required to be paid in ether. These fees are calculated based on the computational resources and the time that is required to execute them, so ether is also called the "digital oil". Like Bitcoin, one can obtain ether by (i) being a miner and verifying network transactions, where every 12 seconds 5 ethers are rewarded to the miner, or (ii) by purchasing from another Ethereum entity.

5.2.4 Gas

As it was mentioned in the above, Ethereum nodes are required to be rewarded for executing the transactions and maintaining the Ethereum blockchain, this is achieved by paying ether for each transaction. The amount of ether in these transaction fees indicate the notion of gas. Every transaction contains a number of operations and there is a precise amount of gas unit associated with each operation. Accordingly, the ultimate amount of gas required for a transaction to be executed is equal to the gas needed for all the operations in that transaction. Each transaction has a corresponding gas price which incentivizes the network nodes to execute it and transactions with higher amount of gas are effectively processed faster by the network.

It is almost impossible for a sender to examine the precise amount of gas needed for the completed execution of the transaction he aims to send to the network, thus there is a gas limit associated with each transaction. Gas limit is the maximum amount of gas that a sender can pay as a transaction fee so that he does not loose all of his funds. By doing so, the transaction is certainly executed by the network nodes and the remaining gas (if there is any) is refunded to the payee.

5.2.5 Mining

Like all other blockchain technologies, Ethereum employs a mining process to ensure a secure and valid decentralized record keeping. To do so, a block is broadcast to the network after it is mined by a miner, then it is checked as valid and appended to the blockchain, if it contains a proof of work of a determined difficulty. The proof of work algorithm that is currently applied by the Ethereum is called *Ethash* [68] and it involves finding a nonce input to the algorithm so that the result is below a certain threshold depending on the difficulty. Note that to maintain 12 second block interval in the Ethereum, the mining difficulty fluctuates.

Miners require to invest high computational resources to perform the proof of work as it contains extremely expensive computations. As a reward, they gain 5 ether for every block they create and mine in addition to total gas consumed by the entire transactions within that block. By doing so, Ethereum provides an incentive for the network nodes (miners) to verify the transactions and maintain a decentralized and immutable ledger of information.

5.3 Design and Implementation

While the previous section focused on theoretical concepts of the Ethereum blckchain, in this section, we will outline how Ghazal* system has been developed and designed around these concepts.

5.3.1 Ghazal* Smart Contract

Our system is hosted on the Ethereum blockchain and is managed by a smart contract written in Solidity language.The smart contract acts as an interface to the blockchain and enables entities to (i) register domain names and (ii) manage these names by binding them to cryptographic keys, transfer and/or auction off their domain names *etc.* This means the same entity that is managing the domain is the one who hosts and manages the certificates.

5.3.2 Ghazal* Smart Contract Entities

The only entities that interact with the current implementation of our system are domain owners. Note that domain owners can be any human or organization or any other party who owns a private key that is associated with an Ethereum address. These entities can register domain names, bind their domain names to DNS records and/or cryptographic keys, transfer their domain names to the other Ethereum accounts, and auction off their domain names.

5.3.3 Solidity

As mentioned in the Section 5.2.2, Ghazal* smart contract is written in a high level programming language called Solidity. Solidity is an object oriented Java-like language which takes the human readable smart contract code and and compiles it into the EVM byte code. Currently, it is a widely used programming language among the Ethereum developers. Below We represent Solidity features that have been used in Ghazal*.

5.3.4 Types

As a high level programming language, Solidity supports different data types such as integers and booleans. In the following sections, we discuss data types of the Solidity language (together with a few examples from the code) that have been used in implementing Ghazal*.

```
struct Domain{
        bytes32 DomainName;
        address DomainOwner;
        uint RegistrationTime;
        bytes32[] TLSKeys;
        bool isValue;
        States state;
        ZoneFileStruct ZoneFile;

      }
```

Code 5.1: Domain struct that stores domain names and their attributes in Ghazal* smart contract.

Structure

Solidity allows developers to to define an struct type with a name and related properties inside of it by using the struct statement. Code 5.1 shows one of the Ghazal*'s struct data types; *Domain* that is used to store domain names.

For each registered domain name, a *Domain* struct is initialized and stores the following attributes:

- *Domain Name*–The domain name that is registered.
- *Domain Owner*– The Ethereum address of the account who registers the domain.
- *Registration Time*– The exact time when the domain is registered.
- *TLSKeys*– A dynamically sized byte array which can store infinite number of public keys. As discussed in the previous chapter, we allow entities to obtain multiple certificates for a domain name they own.
- *IsValue*– This boolean value is set to true once a Domain struct is initialized for a key in mapping which we will further discuss.
- *State*– This is a user defined variable which represents the state of a domain at each moment and will be discussed further while explaining the **enum** type.

```
mapping(bytes32 => Domain) public Domains;
```
Code 5.2: Domains mapping in Ghazal smart contract.

- *ZoneFile*– Each domain owns a zonefile struct that allows the domain owner to add the domain's associated resource records.

Arrays

Array data type is meant to store a group elements. Like other programming languages, there are two types of dynamically sized and fixed sized arrays in Solidity. In Ghazal*, we use **Bytes32[] TLSKeys**– a dynamic array of bytes32 to store as many number as public keys.

Mappings

Mapping is referred to a hash table in Solidity which organizes values based on user defined keys. Mappings allow users to look up an specific value using its key type that he has defined. *Domains* is an example of using mapping data structure in Ghazal* system to store the domain structures (see Code 5.2).

As it can be seen in Code 5.2, we declare a mapping called *Domains* which accepts *Domain name* from bytes32 type as its key, and *domain struct* as the value. By doing so, we are able to look up a domain struct with its corresponding domain name and retrieve all the associated attributes.

Given the fact that uninitialized mapping keys (the domains that are not registered yet) are set to zero by default, we use *isValue* boolean in domain struct; once a domain is registered and initialized in Domains mapping, *isValue* is set to *true*. This allows us to further verify whether an specific domain is in registered state or not.

Mappings vs Arrays. Here we discuss the differences between mappings and arrays and the reason why we select mappings while designing our system. To do so, assume that we are interested to retrieve a value x that is stored in 5.1 array.

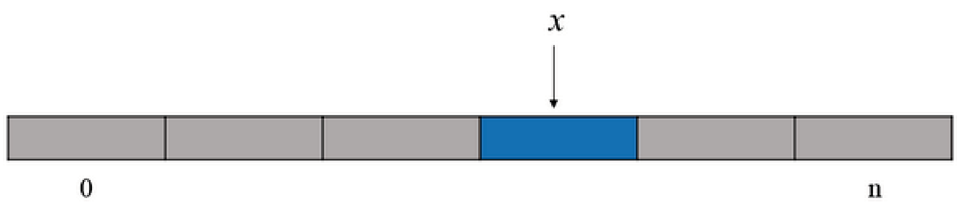


Figure 5.1: Storing value x in an array.

In order to look up the value x, we have to iterate over the entire array as we do not know where is the exact place it is stored. This can be neglected in case of arrays with small size, however, it takes a long time to find an item as the array size grows. On the other hand, in order to store the value x in a mapping, user needs to provide a pair of (key, x), then the key gets hashed and outputs a number which indicates where in the mapping x should be stored (see Figure 5.2). In order

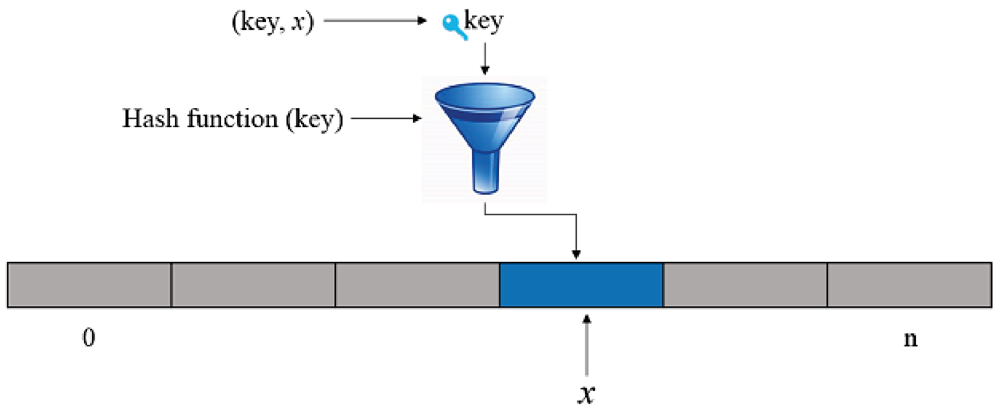


Figure 5.2: Storing value x in a mapping.

to retrieve the value x from a mapping, user needs to provide the associated key, the key gets hashed and supplies the exact location where x is stored in mapping. Note that hash functions are deterministic and they always generate the same hash value for a given input. Therefore, instead of iterating over the entire mapping, one only needs to provide the exact pair of (key, value) every time he wants to look up that value.

According to what have been presented, mappings enjoy a higher level of efficiency in compared with arrays. Retrieving a value from array requires a full scan of the array which results in O(n) complexity, whereas with mapping, Searching for a value is performed in O(1) as it takes a constant time to find that value. Table 5.1 summarizes the average complexity of performing the three *insert*, *search*, and *delete* functions on arrays and mappings data structures. In view of all that has been mentioned so

Function	Arrays	Mappings
Insert	O(1)	O(1)
Search	O(n)	O(1)
Delete	O(n)	O(1)

Table 5.1: Summary of performance characteristics of arrays and mappings.

far, we select mappings to design and implement our system.

Enums

Enums enable smart contract developers to create user-defined types in Solidity. In Ghazal*, we use enums data types to implement state machines and encapsulate functions' behavior based on a given state. Code 5.3 is an example of using enums in Ghazal*. As the code shows, we create a user-defined type called *States* to (i) define every possible states for a domain name and (ii) maintain the states of all domain names that exist within our system.

```
 1 enum States{
 2     Unregistered,
 3     Registered,
 4     Expired,
 5     TLSKeyEntered,
 6     DNSHashEntered,
 7     TLSKey_And_DNSHashEntered}
 8 struct Domain{
 9     bytes32 DomainName;
10         address DomainOwner;
11         uint RegistrationTime;
12         bytes32[] TLSKeys;
13         bool isValue;
14         States state;
15         ZoneFileStruct ZoneFile;}
```

Code 5.3: Defining a new data type called *States* and declaring a new variable called *state* of that type in Ghazal* smart contract.

States contains six values including *unregistered, registered, expired, TLS key entered, DNS has entered, and TLS key_DNS hash entered.* Each of these values indicates a possible state that every domain names may go through. Interestingly, enums can be explicitly converted to/from the any integer types *e.g.,* States(1) represents the *registered* state.

Once the new data type is defined (line 1, Code 5.3), we can declare variables of that type that contain possible stages we have specified. Line 14 of Code 5.3 represents declaring a new variable called *state* as a domain struct's property. Using the *state* variable, we can maintain domain name's possible states which we will further explain how it leads us to design Ghazal* namespace properly.

5.3.5 Functions

Solidity allows developers to define units of code called *functions* in smart contracts and execute those code on EVM. Solidity functions are classified into two types of (i) transactional, also known as functions, and (ii) constant. Transactional functions, as the name implies, generate a transaction to the Ethereum blockchain and can modify the state of a contract, once they are invoked. In contrast, constant functions cannot update the state of the contracts and modify the blockchain. Alternatively, they can be called to return a value to the user who directly calls these function without consuming any amount of gas. Functions can be described with four distinctive visibility marks in Solidity. These include:

- External: These functions are part of the contract definition, although they can not be called and invoked internally and can be merely called by external entities within the blockchain (contracts and/or accounts).
- Public: These functions are default in Solidity and do not need to be determined. Public functions can be accessed and invoked both internally or by external entities within the blockchain (contracts and/or accounts).
- Internal: These functions can merely be accessed and invoked by the contract in which they are defined and its inherited functions and internal libraries.
- Private: These functions can be merely accessed and invoked by the current contract in which they are defined and not by its inherited functions, internal libraries, and external entities within the blockchain (contracts and/or accounts).

5.3.6 Function Modifiers

Function modifiers allow smart contract developers to easily modify the behavior of functions. In solidity, every function can belong to different modifiers, that is, multiple conditions need to be satisfied in order for the function to be executed. At the time of writing the thesis, Ghazal* smart contract contains 11 function modifiers which enforce different conditions, what follows is a description of the five example of these modifiers.

```
1 modifier Costs() {
2         require(msg.value >= Registration_Fee);
3         _;
4     }
```

Code 5.4: Implementation of the *Costs* modifier in Ghazal* smart contract.

```
1 modifier OnlyOwner(bytes32 _DomainName) {
2         require (Domains[_DomainName].DomainOwner == msg.sender);
3         _;
4     }
```

Code 5.5: Implementation of the *OnlyOwner* modifier in Ghazal* smart contract.

Costs **Function Modifier**

As mentioned previously, users can register and/or renew domain names by paying an certain amount of ether as domain registration fee in Ghazal*. To enforce that, the two *register* and *renew* functions are followed by the *costs* modifier which requires a certain amount of fee to be associated with these function calls. Therefore, a user can only invoke and execute these function if he pays the amount that is specified as the registration fee. Additionally, in order to receive ether, these functions are marked *payable*. *costs* modifier first checks the condition prior to executing the function (line 2, Code 5.4) and control flow continues after the "_" in the modifier (line 3, Code 5.4).

OnlyOwner **Function Modifier**

In Ghazal* smart contract, we define the *OnlyOwner* function modifier in a way that it requires a function to be only called from a certain Ethereum address. In fact, any time that a user calls a function on an specific domain name (*e.g.,* add TLS key), the *OnlyOwner* modifier receives the domain name as its argument and allows the user to successfully execute the function only if he is the owner of that domain (see Code 5.5).

```
modifier AtStage(bytes32 _DomainName,States stage_1,States stage_2) {
        require (Domains[_DomainName].state == stage_1 ||     Domains[_DomainName].
            state == stage_2);
        _;
    }
    modifier Not_AtStage(bytes32 _DomainName,States stage_1,States stage_2) {
        require (Domains[_DomainName].state != stage_1 ||     Domains[_DomainName].
            state != stage_2);
        _;
    }
```

Code 5.6: Implementation of the *AtStage* and *Not_AtStage* function modifiers in Ghazal* smart contract.

***AtStage* and *Not_AtStage* Function Modifiers**

In Solidity, functions are atomic operations, that is to say, they can be invoked and executed at any time and any order irrespective of the actual order they are written in the smart contract. Therefore, in order to design interactions within the Ghazal* smart contract accurately, we use *AtStage* and *Not_AtStage* function modifiers to model the states of the contract and prevent incorrect function calls by ensuring that functions can only be executed at certain stages (see Code 5.6). For instance, using the *AtStage* modifier, we enforce the *register(D_i)* function to be executed only if the D_i is at *unregistered* or *expired* state.

***CheckDomainExpiry* Function Modifier**

The *CheckDomainExpiry* function modifier, as the name implies, applies the domains registration expiration after a certain period of time. This function modifier receives a domain name as its argument and changes its state to *expired* if it meets the condition that is specified within the modifier's body (see Code 5.7, line 2)

```
modifier CheckDomainExpiry (bytes32 _DomainName) {
        require (now >= Domains[_DomainName].RegistrationTime + 5 years);
        var DomainVar = Domains[_DomainName];
        DomainVar.state = States.Expired;
        Domains[_DomainName] = DomainVar;
        _;
    }
```

Code 5.7: Implementation of the *CheckDomainExpiry* function modifier in Ghazal* smart contract.

5.4 Evaluation

So far this chapter has focused the Ethereum concepts and data structures in addition to Ghazal* smart contract's design and description of its components. The aim of Section 5.4 is to provide the technical implementation details of our system on the Ethereum blockchain. We specifically discuss the costs related to the deployment of Ghazal* smart contract on the Ethereum blockchain in addition to executing its functions on the Ethereum virtual machine. Moreover, a smart contract analysis tool is used to analyze the security of our system against a several number of security threats to which smart contracts are often vulnerable.

5.4.1 Deployment

In order to analyze the Ghazal* smart contract in today's Ethereum blockchain, Ropsten, Ethereum test network, has been used [69]. Test networks replicate the Ethereum network and EVM as well as offering an inexpensive way for smart contract developers to test their codes. Ropsten is a public test network that entirely stimulates the Ethereum peer to peer network except that it provides free gas. Therefore, we successfully tested Ghazal* smart contract on this test network without paying the real cost of gas for our executions.

5.4.2 Gas Estimation

Ghazal smart contract is implemented in 370 lines of Solidity language, a high level programming language resembles to JavaScript, and tested on the Ethereum test network. We use the Solidity compiler to evaluate the rough cost for publishing the Ghazal* smart contract on the Ethereum blockchain as well as the cost for the various operations to be executed on the Ethereum virtual machine. As of January 2018, 1 gas $= 21 \times 10^{-9}$ ether[1], and 1 ether = \$882.92[2].

Table 5.2 represents the estimated costs for Ghazal* (and its inherited Ghazal functionality) smart contract deployment and function invocation in both gas and USD. As it can be seen from both Table 1, the most considerable cost is the one-time cost paid to deploy the system on Ethereum. There are then relatively small costs associated with executing the functions, *i.e.,* users could easily register a domain by paying \$3.15 or they could bind a key to the domain they own for a cost of \$1.43, which is relatively cheap when compared with the real world costs associated with these operations.

5.4.3 Security Analysis

Ethereum smart contracts, in particular the ones implemented in Solidity, are notorious for programming pitfalls. As they generally transfer and handle assets of considerable value, bugs in Solidity code could result in serious vulnerabilities which can be exploited by adversaries. We use standard defensive programming approaches, in particular around functions that transfer money (such as the auction function that refunds the security deposits), by using explicitly coded state machines and locks, and by not making state-changes after transfers. We also analyze Ghazal* against

[1]https://ethstats.net/
[2]https://coinmarketcap.com/

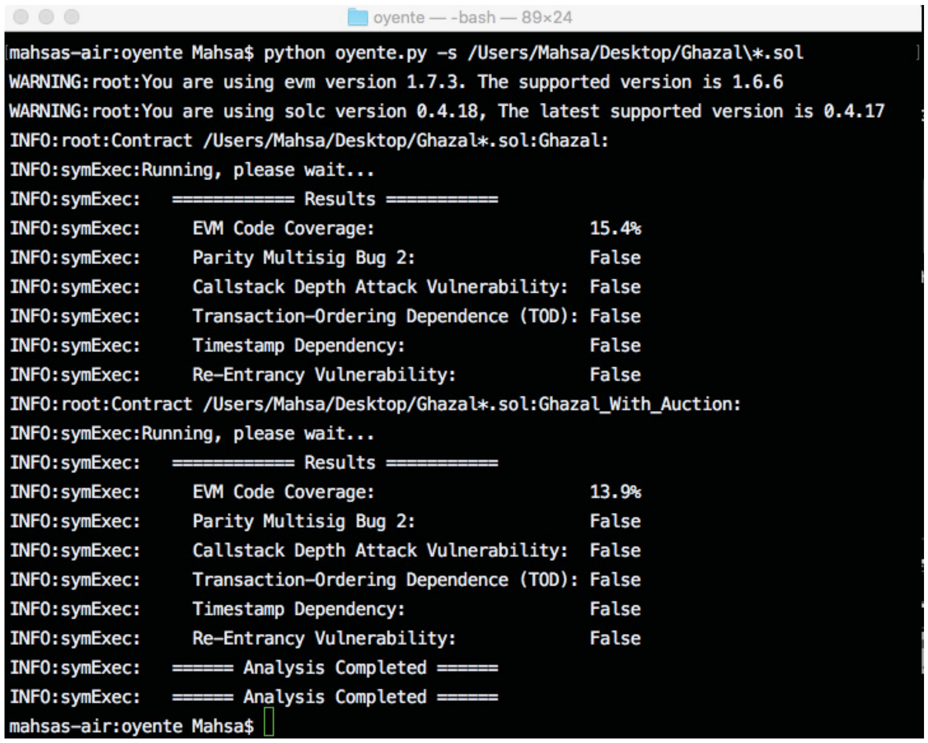
Operation	**Gas**	**Gas Cost in Ether**	**Gas Cost in USD**
Register	169 990	3.56×10^{-3}	\$3.15
Renew	54 545	1.14×10^{-3}	\$1.01
Transfer_Domain	53 160	1.11×10^{-3}	\$0.98
Add_TLSKey	77 625	1.63×10^{-3}	\$1.43
Add_ZoneFile	57 141	1.19×10^{-3}	\$1.05
Add_TLSKey_AND_ZoneFile	68 196	1.43×10^{-3}	\$1.26
Revoke_TLSkey	37 672	7.91×10^{-4}	\$0.69
StartAuction	119 310	2.50×10^{-3}	\$2.21
Bid	112 491	2.36×10^{-3}	\$2.08
Withdraw_bids	46 307	9.72×10^{-4}	\$0.85
Withdraw_deposits	47 037	9.87×10^{-4}	\$0.87
Settle	77 709	1.63×10^{-3}	\$1.44
Ghazal* Contract Creation	2 402 563	0.05	\$44.54

Table 5.2: Gas used for operations in the Ghazal* smart contract.

Oyente, a symbolic execution tool proposed by Luu *et al.* [1] which looks for potential security bugs like the re-entry attack (infamously). The results of the security analysis represent that both of the smart contracts are not vulnerable to any known critical security issue (see Figure 5.3).

5.5 Conclusion

In this chapter, we described and discussed the main concepts of the Ethereum blockchain as well as the Solidity language. We also provided a few code snippets of Ghazal* to show a better understanding of our system. In the last section, a thorough security and cost evaluation of Ghazal* was performed. According to the results, our system is totally secure against the existing security vulnerabilities. The overall system costs under \$100 to deploy Basic actions like domain registration costs under \$5.



oyente — -bash — 89×24

```
mahsas-air:oyente Mahsa$ python oyente.py -s /Users/Mahsa/Desktop/Ghazal\*.sol
WARNING:root:You are using evm version 1.7.3. The supported version is 1.6.6
WARNING:root:You are using solc version 0.4.18, The latest supported version is 0.4.17
INFO:root:Contract /Users/Mahsa/Desktop/Ghazal*.sol:Ghazal:
INFO:symExec:Running, please wait...
INFO:symExec:   ============ Results ===========
INFO:symExec:     EVM Code Coverage:                     15.4%
INFO:symExec:     Parity Multisig Bug 2:                 False
INFO:symExec:     Callstack Depth Attack Vulnerability:  False
INFO:symExec:     Transaction-Ordering Dependence (TOD): False
INFO:symExec:     Timestamp Dependency:                  False
INFO:symExec:     Re-Entrancy Vulnerability:             False
INFO:root:Contract /Users/Mahsa/Desktop/Ghazal*.sol:Ghazal_With_Auction:
INFO:symExec:Running, please wait...
INFO:symExec:   ============ Results ===========
INFO:symExec:     EVM Code Coverage:                     13.9%
INFO:symExec:     Parity Multisig Bug 2:                 False
INFO:symExec:     Callstack Depth Attack Vulnerability:  False
INFO:symExec:     Transaction-Ordering Dependence (TOD): False
INFO:symExec:     Timestamp Dependency:                  False
INFO:symExec:     Re-Entrancy Vulnerability:             False
INFO:symExec:   ====== Analysis Completed ======
INFO:symExec:   ====== Analysis Completed ======
mahsas-air:oyente Mahsa$
```

Figure 5.3: Results of Ghazal* security analysis using Oyente [1].

Chapter 6

Concluding Remarks

In this thesis, we took a deep look into the public key infrastructure and the traditional web certification model. A thorough empirical study of the CA ecosystem has been performed and the major issues were entirely discussed. We introduce a novel uni-authoritative PKI and naming system called Ghazal* on the Ethereum blockchain. We hope that uni-authoritative systems with programmability continue to be explored in the literature. There are many open problems to work on. First and foremost is understanding the scalability issues and how to minimize the amount of data a client browser needs to fetch for each domain lookup. Blockstack has done an excellent job on this issue for non-programmable contracts. Future work could also look at the layer above the smart contract: building web tools with user interfaces to enable interaction with the underlying functions. Finally, while auctions are one illustrative example of why programmability might be added to a PKI, we are sure there are many others. The modular design of Ghazal* using object-oriented programming should allow easy additions to our base contract, which we will provide as open source. Indeed, the auction itself in Ghazal* was added via inheritance and one function override (to enforce that ownership transfers, part of the parent class, could not be called during a live auction).

Bibliography

[1] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 254–269, ACM, 2016.

[2] C. M. Ellison, "The nature of a useable pki," *Computer Networks*, vol. 31, no. 8, pp. 823–830, 1999.

[3] R. Perlman, "An overview of pki trust models," *IEEE network*, vol. 13, no. 6, pp. 38–43, 1999.

[4] R. Housley, W. Polk, W. Ford, and D. Solo, "Internet x. 509 public key infrastructure certificate and certificate revocation list (crl) profile," tech. rep., 2002.

[5] M. Wenbo, "Modern cryptography: theory and practice," *Publisher: Prentice Hall PTR, Copyright: Hewlett Packard*, 2004.

[6] C. Adams and S. Lloyd, *Understanding PKI: concepts, standards, and deployment considerations*. Addison-Wesley Professional, 2003.

[7] J. R. Vacca, *Public key infrastructure: building trusted applications and Web services*. CRC Press, 2004.

[8] P. Eckersley and J. Burns, "An observatory for the ssliverse," *Talk at Defcon*, vol. 18, 2010.

[9] "Comodo report of incident - comodo detected and thwarted an intrusion on 26-mar-2011." https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html. (Accessed on 07/11/2017).

[10] D. Kaminsky, L. Sassaman, and M. Patterson, "Pki layer cake: New collision attacks against the global x. 509 ca infrastructure. black hat usa, august 2009."

[11] "Google online security blog: Enhancing digital certificate security." https://security.googleblog.com/2013/01/enhancing-digital-certificate-security.html. (Accessed on 07/11/2017).

[12] "Google online security blog: Further improving digital certificate security." https://security.googleblog.com/2013/12/further-improving-digital-certificate.html. (Accessed on 07/11/2017).

[13] "Google, yahoo, skype targeted in attack linked to iran - cnet." https://www.cnet.com/news/google-yahoo-skype-targeted-in-attack-linked-to-iran/. (Accessed on 07/11/2017).

[14] J. Clark and P. v. Oorschot, "SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements," in *IEEE S&P*, 2013.

[15] H. A. Kalodner, M. Carlsten, P. Ellenbogen, J. Bonneau, and A. Narayanan, "An empirical study of namecoin and lessons for decentralized namespace design.," in *WEIS*, 2015.

[16] C. Fromknecht, D. Velicanu, and S. Yakoubov, "Certcoin: A namecoin based decentralized authentication system 6.857 class project," 2014.

[17] L. Axon and M. Goldsmith, "Pb-pki: a privacy-aware blockchain-based pki," 2016.

[18] M. Ali, J. C. Nelson, R. Shea, and M. J. Freedman, "Blockstack: A global naming and storage system secured by blockchains.," in *USENIX Annual Technical Conference*, pp. 181–194, 2016.

[19] B. Laurie, "Certificate transparency," *Queue*, vol. 12, no. 8, p. 10, 2014.

[20] "git.eff.org git - sovereign-keys.git/blob - sovereign-key-design.txt." `https://git.eff.org/?p=sovereign-keys.git;a=blob;f=sovereign-key-design.txt;hb=HEAD`. (Accessed on 01/10/2018).

[21] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski, "Arpki: Attack resilient public-key infrastructure," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 382–393, ACM, 2014.

[22] S. Matsumoto and R. M. Reischuk, "Ikp: Turning a pki around with blockchains.," *IACR Cryptology ePrint Archive*, vol. 2016, p. 1018, 2016.

[23] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "Coniks: Bringing key transparency to end users.," in *USENIX Security Symposium*, pp. 383–398, 2015.

[24] J. Bonneau, "Ethiks: Using ethereum to audit a coniks key transparency log," in *International Conference on Financial Cryptography and Data Security*, pp. 95–105, Springer, 2016.

[25] B. Kulynych, M. Isaakidis, C. Troncoso, and G. Danezis, "Claimchain: Decentralized public key infrastructure," *arXiv preprint arXiv:1707.06279*, 2017.

[26] T. Hardjono and A. S. Pentland, "Verifiable anonymous identities and access control in permissioned blockchains."

[27] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping authorities" honest or bust" with decentralized witness cosigning," in *Security and Privacy (SP), 2016 IEEE Symposium on*, pp. 526–545, Ieee, 2016.

[28] M. Chase and S. Meiklejohn, "Transparency overlays and applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 168–179, ACM, 2016.

[29] M. Al-Bassam, "Scpki: A smart contract-based pki and identity system," in *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, pp. 35–40, ACM, 2017.

[30] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[31] C. P. Pfleeger and S. L. Pfleeger, *Security in computing.* Prentice Hall Professional Technical Reference, 2002.

[32] B. Schneier, *Applied cryptography: protocols, algorithms, and source code in C.* john wiley & sons, 2007.

[33] A. Arsenault and S. Turner, "Internet x. 509 public key infrastructure: Roadmap," *PKIX Working Group Internet Draft*, pp. 1–55, 2002.

[34] E. Rescorla, *SSL and TLS: designing and building secure systems*, vol. 1. Addison-Wesley Reading, 2001.

[35] E. Rescorla, "Http over tls," 2000.

[36] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[37] L. Lamport, R. Shostak, and M. Pease, “The byzantine generals problem,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[38] L. Lamport, “The part-time parliament,” *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[39] M. Jakobsson and A. Juels, “Proofs of work and bread pudding protocols,” in *Secure Information Networks*, pp. 258–272, Springer, 1999.

[40] V. Buterin *et al.*, “A next-generation smart contract and decentralized application platform,” *white paper*, 2014.

[41] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum Project Yellow Paper*, vol. 151, 2014.

[42] “Ethereum development tutorial ethereum/wiki wiki.” `https://github.com/ethereum/wiki/wiki/Ethereum-Development-Tutorial`. (Accessed on 07/12/2017).

[43] N. Szabo, “Formalizing and securing relationships on public networks,” *First Monday*, vol. 2, no. 9, 1997.

[44] C. Soghoian and S. Stamm, “Certified lies: Detecting and defeating government interception attacks against ssl (short paper),” in *International Conference on Financial Cryptography and Data Security*, pp. 250–259, Springer, 2011.

[45] X. d. C. de Carnavalet and M. Mannan, “Killed by proxy: Analyzing client-end tls interception software,” in *Network and Distributed System Security Symposium*, 2016.

[46] "Alternative methods of domain control validation (dcv) - powered by kayako help desk software." https://support.comodo.com/index.php?/Knowledgebase/Article/View/791/0/alternative-methods-of-domain-control-validation-dcv. (Accessed on 10/16/2017).

[47] "Blackhat-dc-09-marlinspike-defeating-ssl.pdf." http://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf. (Accessed on 10/16/2017).

[48] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, "Analysis of the https certificate ecosystem," in *Proceedings of the 2013 conference on Internet measurement conference*, pp. 291–304, ACM, 2013.

[49] "Ssl certificates rapid7/sonar wiki github." https://github.com/rapid7/sonar/wiki/SSL-Certificates. (Accessed on 10/23/2017).

[50] "More ssl certificates rapid7/sonar wiki github." https://github.com/rapid7/sonar/wiki/More-SSL-Certificates. (Accessed on 10/23/2017).

[51] J. A. Halderman, "Fast internet-wide scanning and its security applications,"

[52] B. Laurie, A. Langley, and E. Kasper, "Certificate transparency," tech. rep., 2013.

[53] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, "Analysis of the https certificate ecosystem," in *IMC*, 2013.

[54] R. Holz, L. Braun, N. Kammenhuber, and G. Carle, "The SSL landscape: A thorough analysis of the X.509 PKI using active and passive measurements," in *IMC*, 2011.

[55] M. Zusman, “Criminal charges are not pursued: Hacking pki,” *DEFCON 17*, 2009.

[56] “Godaddy owns up to role in epic twitter account hijacking — pcworld.” https://www.pcworld.com/article/2093100/godaddy-owns-up-to-role-in-twitter-account-hijacking-incident.html. (Accessed on 02/13/2018).

[57] S. Son and V. Shmatikov, “The hitchhikers guide to dns cache poisoning,” *Security and Privacy in Communication Networks*, pp. 466–483, 2010.

[58] D. Liu, S. Hao, and H. Wang, “All your dns records point to us: Understanding the security threats of dangling dns records,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1414–1425, ACM, 2016.

[59] “Home.” http://www.ethereum-alarm-clock.com/. (Accessed on 12/29/2017).

[60] M. Myers, “Revocatoin: Options and challenges,” in *Financial Cryptography*, pp. 165–171, Springer, 1998.

[61] E. Topalovic, B. Saeta, L.-S. Huang, C. Jackson, and D. Boneh, “Towards short-lived certificates,” *Web 2.0 Security and Privacy*, 2012.

[62] D. Wendlandt, D. G. Andersen, and A. Perrig, “Perspectives: Improving SSH-style host authentication with multi-path probing,” in *USENIX Annual Tech*, 2008.

[63] M. Marlinspike, “SSL and the future of authenticity,” in *Black Hat USA*, 2011.

[64] “Mutan ethereum/go-ethereum wiki github.” https://github.com/ethereum/go-ethereum/wiki/Mutan. (Accessed on 11/06/2017).

[65] "Lll poc 6 ethereum/cpp-ethereum wiki github." https://github.com/ethereum/cpp-ethereum/wiki/LLL-PoC-6/7a575cf91c4572734a83f95e970e9e7ed64849ce. (Accessed on 11/06/2017).

[66] "Serpent ethereum/wiki wiki github." https://github.com/ethereum/wiki/wiki/Serpent. (Accessed on 11/06/2017).

[67] "Solidity solidity 0.4.19 documentation." http://solidity.readthedocs.io/en/latest/. (Accessed on 11/06/2017).

[68] "Ethash ethereum/wiki wiki github." https://github.com/ethereum/wiki/wiki/Ethash. (Accessed on 11/06/2017).

[69] "Github - ethereum/ropsten: Ropsten public testnet pow chain." https://github.com/ethereum/ropsten. (Accessed on 11/20/2017).

Appendix A

Ghazal Full Smart Contract

```
pragma solidity ^0.4.0;
contract Ghazal{
    bytes32 public DomainName;
    uint Registration_Fee = 1 ether; //The amount that a user has to pay in order to register a domain.
    mapping (bytes32 => Domain) internal Domains;
    Domain internal CurrentDomain;
    enum States {Unregistered,Registered,Expired,TLSKeyEntered,ZoneFileEntered, TLSKey_And_ZoneFileEntered}

    mapping(address => uint) internal refunds;

//
    ***********************************************************************************************

    //Each domain will hvae a zonefile struct in which the domain owner adds the domain name associated resource records.
    // for now we only include the IP address.
    struct ZoneFileStruct{
       string IP_Address;
    }
//
    ***********************************************************************************************

   struct Domain{ //Domain struct represents each domain which posses DomainName,
      RegistrantName, Validity
        bytes32 DomainName;
        address DomainOwner;
        uint RegistrationTime;
        bytes32[] TLSKeys; // Dynamic array of bytes32 to store multiple
            certificates for a single domain.
        bool isValue; //IF the Domain struct is initiallized for a key (_DomianName)
            , this value is set to true.
        States state; //Keeps the state of the domain.
        ZoneFileStruct ZoneFile;

      }
//
    ***********************************************************************************************

    //Cost function modifier allows a function to get executed if the msg.value is
        equal or greater than the Registration_Fee (Which we defined as 1 ether)
    modifier Costs() {
        require(msg.value >= Registration_Fee);
        _;
      }
//
    ***********************************************************************************************

    //OnlyOwner function modifier allows a function to get executed if the entity
        that is invoking a function is the same as domain owner.
    modifier OnlyOwner(bytes32 _DomainName) {
        require(Domains[_DomainName].DomainOwner == msg.sender);
        _;
    }
//
    ***********************************************************************************************

    //AtStage function modifier allows a function to get executed if the domain is
        in desired states.
    modifier AtStage(bytes32 _DomainName, States stage_1, States stage_2) {
        require (Domains[_DomainName].state == stage_1  || Domains[_DomainName].
            state == stage_2);
        _;
    }
```

```
//*************************************************************************************************
    //Not_AtStage function modifier allows a function to get executed if the domain is not in the specified states.
    modifier Not_AtStage(bytes32 _DomainName, States stage_1, States stage_2) {
        require (Domains[_DomainName].state != stage_1 && Domains[_DomainName].state != stage_2);
        _;
    }
//*************************************************************************************************
    //CheckDomainExpiry function modifier checks if the domain name is expired or not.
    modifier CheckDomainExpiry (bytes32 _DomainName) {
        if (Domains[_DomainName].isValue == false) {Domains[_DomainName].state= States.Unregistered;} //IF the Domain struct is initiallized for the key (_DomianName), it updates the domain's state to Unregistered.
        if (now >= Domains[_DomainName].RegistrationTime + 10 minutes) {Domains[_DomainName].state = States.Expired;} // each domain expires in 5 years.
        _;
    }
//*************************************************************************************************
    //A user can Register a Domain using the Register function.
    function Register (string _DomainName)  payable public CheckDomainExpiry (stringToBytes32(_DomainName)) Costs() AtStage(stringToBytes32(_DomainName), States.Unregistered,States.Expired)
    {
        DomainName = stringToBytes32(_DomainName);
        CurrentDomain.DomainName = DomainName;
        CurrentDomain.DomainOwner = msg.sender;
        CurrentDomain.RegistrationTime = now;
        CurrentDomain.isValue = true;
        delete CurrentDomain.TLSKeys;
        CurrentDomain.state = States.Registered;
        Domains[DomainName] = CurrentDomain;
        refunds[block.coinbase] += Registration_Fee;
        uint refund = refunds[block.coinbase];
        refunds[block.coinbase] = 0;
        block.coinbase.transfer(refund);

    }
//****************************************************************************************************
    //Domain Owner can renew the domain at least 1 year before the domain is expired. Note that Domian validation period is 5 years.
    function Renew (string _DomainName) public payable CheckDomainExpiry (stringToBytes32(_DomainName)) Costs() OnlyOwner(stringToBytes32(_DomainName))
    {
        DomainName = stringToBytes32(_DomainName);
        require (now >= Domains[DomainName].RegistrationTime + 10 minutes);
        Domains[DomainName].RegistrationTime = now;
        refunds[block.coinbase] += Registration_Fee;
        uint refund = refunds[block.coinbase];
        refunds[block.coinbase] = 0;
        block.coinbase.transfer(refund);
    }
//****************************************************************************************************
    //A user can add unlimited number of certificates to his Domain using the Add_TLSKey function.
    function Add_TLSKey (string _DomainName,bytes32 _TLSKey)  public CheckDomainExpiry (stringToBytes32(_DomainName)) Not_AtStage(stringToBytes32(_DomainName),States.Unregistered,States.Expired) OnlyOwner(stringToBytes32(_DomainName))
    {
        DomainName = stringToBytes32(_DomainName);
        Domains[DomainName].TLSKeys.push(_TLSKey);
        if (Domains[DomainName].state == States.Registered) {Domains[DomainName].state = States.TLSKeyEntered;}//if the domain is in the registered state, it transitions to TLSKeyEntered.
        if (Domains[DomainName].state == States.ZoneFileEntered) {Domains[DomainName].state = States.TLSKey_And_ZoneFileEntered;}//if the domain contains the DNSHash, it transitions to TLSKey_And_DNSHashEntered.
        //if Domain's state is TLSKeyEntered OR TLSKey_And_DNSHashEntered, its state will not  change.
    }

//*****************************************************************************************
    //A user can add the hash of it DNS to his Domain using the Add_DNSHash function.
    function Add_Zonefile (string _DomainName,bytes32 _Zone_Hash, string _IP_Address)  public CheckDomainExpiry (stringToBytes32(_DomainName)) Not_AtStage(stringToBytes32(_DomainName),States.Unregistered,States.Expired) OnlyOwner(stringToBytes32(_DomainName))
    {
        DomainName = stringToBytes32(_DomainName);
        //Domains[DomainName].DNSHash = _DNSHash;
        Domains[DomainName].ZoneFile.IP_Address = _IP_Address;
        if (Domains[DomainName].state == States.Registered) {Domains[DomainName].state = States.ZoneFileEntered;}//if the domain is in the registered state, it transitions to DNSHashEntered.
        if (Domains[DomainName].state == States.TLSKeyEntered) {Domains[DomainName].state = States.TLSKey_And_ZoneFileEntered;}//if the domain contains the TLSKey, it transitions to TLSKey_And_DNSHashEntered.
        //if Domain's state is DNSHashEntered OR TLSKey_And_DNSHashEntered, its state will not  change.
    }
//*****************************************************************************************
    //A user can add certificates and DNSHash to his domain usign the Add_TLSKey_AND_DNSHash function.
    function Add_TLSKey_AND_Zonefile (string _DomainName,bytes32 _TLSKey, bytes32 _Zone_Hash, string _IP_Address)  public CheckDomainExpiry (stringToBytes32(_DomainName)) Not_AtStage(stringToBytes32(_DomainName),States.Unregistered,States.Expired) OnlyOwner(stringToBytes32(_DomainName))
    {
        DomainName = stringToBytes32(_DomainName);
        Domains[DomainName].ZoneFile.IP_Address = _IP_Address;
        Domains[DomainName].TLSKeys.push(_TLSKey);
        Domains[DomainName].state = States.TLSKey_And_ZoneFileEntered;
    }
//***********************************************************************************************
    //DomainOwner can transfer the Domain to any address he wants if and only if the Domain is not Unregistered and Expired.
    //1- DomainOwner can only transfer the domain name. To do so, he wipes the associated DNS Hash and TLS Key by supplying them with zero.
    //2-  DomainOwner can transfer the domain name in addition to the corresponding certificate and and/or DNS Hash. This is done by supplying these arguments with their previous values.
    //Note that the Domain State and Registration_Time will not change and remain the same .
    function Transfer_Domain (string _DomainName, address _Reciever,bytes32 _TLSKey, string _IP_Address) public CheckDomainExpiry (stringToBytes32(_DomainName)) Not_AtStage(stringToBytes32(_DomainName),States.Unregistered,States.Expired) OnlyOwner(stringToBytes32(_DomainName))
    {
        DomainName = stringToBytes32(_DomainName);
        Domains[DomainName].DomainOwner = _Reciever;
        if (_TLSKey == 0 && stringToBytes32(_IP_Address) != 0) { Wipe_TLSKeys(DomainName); }
        if (stringToBytes32(_IP_Address) == 0 && _TLSKey != 0 ) {  Wipe_IP_address(DomainName); }
        if (stringToBytes32(_IP_Address) == 0 && _TLSKey == 0 ) { Wipe_TLSKeys_and_IP_address(DomainName); }
    }
//***********************************************************************************************
    function Wipe_TLSKeys (bytes32 _DomainName) internal{
      delete Domains[_DomainName].TLSKeys;
      if (Domains[_DomainName].state == States.TLSKey_And_ZoneFileEntered) {Domains[_DomainName].state = States.ZoneFileEntered;}
    }
//***********************************************************************************************
    function Wipe_IP_address (bytes32 _DomainName) internal{
      delete Domains[_DomainName].ZoneFile.IP_Address;
      if (Domains[_DomainName].state == States.TLSKey_And_ZoneFileEntered) {Domains[_DomainName].state = States.TLSKeyEntered;}
    }
//***********************************************************************************************
    function Wipe_TLSKeys_and_IP_address (bytes32 _DomainName) internal{
      delete Domains[_DomainName].ZoneFile.IP_Address;
      delete Domains[_DomainName].TLSKeys;
      Domains[_DomainName].state = States.Registered;
    }
//***********************************************************************************************
    //A user can revoke any certificates that belong to his domain using the Revoke_TLSkey function.
    function Revoke_TLSkey (string _DomainName, bytes32 _TLSKey) public CheckDomainExpiry (stringToBytes32(_DomainName)) Not_AtStage(stringToBytes32(_DomainName),States.Unregistered,States.Expired) OnlyOwner(stringToBytes32(_DomainName))
    {
      DomainName = stringToBytes32(_DomainName);
      for (uint j=0; j<Domains[DomainName].TLSKeys.length;j++)
      {
        if (Domains[DomainName].TLSKeys[j] == _TLSKey){ delete Domains[DomainName].TLSKeys[j]; }
      }
    }
//***********************************************************************************************
    //stringToBytes32 is an internal function which converts bytes to string whenever called.
    function stringToBytes32(string memory source) internal pure returns (bytes32 result)
    {
    bytes memory tempEmptyStringTest = bytes(source);
    if (tempEmptyStringTest.length == 0) {
        return 0x0;
    }
    assembly {
        result := mload(add(source, 32))
    }
}
//
    ***********************************************************************************************

    //Get_TLSKey is a constant function which returns the TLSKeys a domain name.
    function Get_TLSKey (string _DomainName) public view returns (bytes32[])
    {
      var DomainVar = Domains[stringToBytes32(_DomainName)];
      return DomainVar.TLSKeys;
    }
//
    ***********************************************************************************************

    function Set_Auction_Result (bytes32 _DomainName) internal
    {
      Wipe_TLSKeys_and_IP_address(_DomainName);
      Domains[_DomainName].DomainOwner = msg.sender;
    }

//
    ***********************************************************************************************

}
//*******************************************AUCTION SMART CONTRACT
    ************************************

contract Ghazal_With_Auction is Ghazal{
    enum Stages {UnInitiallized,Opened, Locked, Ended} //Opened: biddingTime,
        Settlement has not yet started.
                                                       //Locked: biddingTime's over,
                                                            Settlement's satrted and
                                                            not finished yet.
                                                       //Ended: biddingTime and
                                                           Settlement are both over,
                                                            which means the auction'
                                                           s ended.

    //There is an struct called "AuctionStruct" for each auction that will be invoked.
    struct AuctionStruct
    {
        uint CreationTime;      //The time auction was opened.
        address Owner;          //The address who opened the auction.
        uint highestBid;        //The highestBid that has been bid in the auction.
        address highestBidder;  //The address who bid the highest bid in the auction.
        address Winner;         //The address of the winner of this auction.
        Stages stage;           //variable stage is frm type Stages which keeps the stage of the auction.
        mapping(address => uint) pendingReturns;    //To return the bids that were overbid.
        mapping(address => uint) deposits;          //To return the deposits they've made.
        mapping(address => bool) already_bid;       //Once an address bids in the auction this variable will be set to true
        bool AuctionisValue;                        //So the next time they bid in the same auction, they dont have to deposit again.
      }

    //AuctionLists mappings store auction structs, the keys are the DomainNames that are auctioned and the values are the auction structs.
    mapping (bytes32 => AuctionStruct) internal AuctionLists;

    uint Deposit_Fee = 1 ether;
    uint public biddingTime = 4 minutes;   //Bidding period. Users can ONLY bid in this period.
    uint public Settlement = 4 minutes;    //Settlement period. Users can Withdraw their pendingReturns (bids that were overbid)
                                           //Plus their deposits. Note that the Winner can withdraw his deposit only if the auction is ended and he claims the Domain
                                           //and pays to the DomainOwner.

//
    ****************************************************************************************************
    modifier OnlyWinner(bytes32 _DomainName) {
        require (AuctionLists[_DomainName].Winner == msg.sender && AuctionLists[_DomainName].stage != Stages.Opened);
        _;
    }

// ***********************************************************************************************

    //Checks if the auction's state.
    modifier CheckAuctionStage (bytes32 _DomainName) {
        if (AuctionLists[_DomainName].AuctionisValue == false) {AuctionLists[_DomainName].stage = Stages.UnInitialized;}
        if (now >= AuctionLists[_DomainName].CreationTime + biddingTime + Settlement ) {AuctionLists[_DomainName].stage = Stages.Ended;}
        if (now >= AuctionLists[_DomainName].CreationTime + biddingTime && now <= AuctionLists[_DomainName].CreationTime + biddingTime + Settlement) {AuctionLists[_DomainName].stage = Stages.Locked;} // each domain expires in 5 years.
        _;

    }
// ***********************************************************************************************

    modifier AuctionAtStage(bytes32 _DomainName, Stages stage_1, Stages stage_2 ) {
        require (AuctionLists[_DomainName].stage == stage_1 || AuctionLists[_DomainName].stage == stage_2);
        _;
    }
// ***********************************************************************************************

    modifier ToBidAuctionAtStage(bytes32 _DomainName, Stages stage_1) {
        require (AuctionLists[_DomainName].stage == stage_1);
        _;
    }
// ***********************************************************************************************
```

```
    modifier DomainNotAtStage(bytes32 _DomainName) {
        require (Domains[_DomainName].state != States.Expired && now <= Domains[
            _DomainName].RegistrationTime + 10 minutes - biddingTime - Settlement);
        _;
    }
//
    ***********************************************************************************************

    modifier NotWinner(bytes32 _DomainName) {
        require (AuctionLists[_DomainName].Winner != msg.sender);
        _;
    }
//
    ***********************************************************************************************

    //To start and auction on a DomainName.
    function StartAuction(string _DomainName) public DomainNotAtStage (
        stringToBytes32(_DomainName)) CheckAuctionStage(stringToBytes32(_DomainName)
        ) OnlyOwner(stringToBytes32(_DomainName)) AuctionAtStage(stringToBytes32(
        _DomainName),Stages.Ended,Stages.UnInitiallized)

    {   //1-Only the DomainOwner call open auction on a domain.
        //2-There should not be any other auction currently open on the same domain.
        //3-Domain expiration should be greater than the whole period of auction (
            biddingTime+Settlement)

        var Domainname = stringToBytes32(_DomainName);
        AuctionLists[Domainname].Owner = msg.sender;
        AuctionLists[Domainname].CreationTime = now;
        AuctionLists[Domainname].stage = Stages.Opened;
        AuctionLists[Domainname].AuctionisValue = true;
    }
//
    ***********************************************************************************************

    //To bid in the auction, ONLY when the auction is Opened.
    function Bid(string _DomainName) payable public CheckAuctionStage(
        stringToBytes32(_DomainName))  ToBidAuctionAtStage(stringToBytes32(
        _DomainName),Stages.Opened)
    {
        var Domainname = stringToBytes32(_DomainName);
        uint  bid;

        //If the bidder has already bid in this auction, he does not deposit.
        if (AuctionLists[Domainname].already_bid[msg.sender] == true) {bid = msg.value;}
        else
        {
            bid = msg.value-Deposit_Fee;  //If the bidder has NOT bid in this auction, he deposits.
            AuctionLists[Domainname].deposits[msg.sender] = Deposit_Fee; //Deposit_Fee will be added to the bidder's deposits.
            AuctionLists[Domainname].already_bid[msg.sender] = true;
        }

        //if the bidder's bid is not higher than the highest bid, send the money back.
        //By adding the bids of the person to his pending returns which he can withdrwa when the auction is Locked.
        require(bid> AuctionLists[Domainname].highestBid);

        if (AuctionLists[Domainname].highestBidder != 0)
        {
            AuctionLists[Domainname].pendingReturns[AuctionLists[Domainname].highestBidder] +=  AuctionLists[Domainname].highestBid;
        }
        AuctionLists[Domainname].highestBidder = msg.sender;
        AuctionLists[Domainname].highestBid = bid;
        AuctionLists[Domainname].Winner =  AuctionLists[Domainname].highestBidder;
    }

//***********************************************************************************************

// Withdraw a bid that was overbid. Only when Auction is Locked.
    function withdraw_bid(string _DomainName) public  CheckAuctionStage(stringToBytes32(_DomainName)) ToBidAuctionAtStage(stringToBytes32(_DomainName),Stages.Locked)returns (bool)
    {
        var Domainname = stringToBytes32(_DomainName);
        uint amount = AuctionLists[Domainname].pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the recipient
            // can call this function again as part of the receiving call
            // before ‘send‘ returns.
            AuctionLists[Domainname].pendingReturns[msg.sender] = 0;

            if (!msg.sender.send(amount)) {
                // No need to call throw here, just reset the amount owing
                AuctionLists[Domainname].pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }
//
    **********************************************************************************************
// Withdraw Deposits. Only when Auction is Locked. Th Winner CANNOT witdraw his
    deposit.
    function Withdraw_deposits(string _DomainName) public CheckAuctionStage(
        stringToBytes32(_DomainName)) NotWinner (stringToBytes32(_DomainName))
        ToBidAuctionAtStage(stringToBytes32(_DomainName),Stages.Locked) returns (
        bool) {

        var Domainname = stringToBytes32(_DomainName);
        uint amount = AuctionLists[Domainname].deposits[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the recipient
            // can call this function again as part of the receiving call
            // before ‘send‘ returns.
            AuctionLists[Domainname].deposits[msg.sender] = 0;

            if (!msg.sender.send(amount)) {
                // No need to call throw here, just reset the amount owing
                AuctionLists[Domainname].deposits[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

//    ***********************************************************************************************
//End the auction,send the highest bid to the auction's owner, transfer the Domainname to the auction's winner.
//Only Winner acn call the function Settle. and ONLY when the auction is ended.
    function Settle(string _DomainName) public CheckAuctionStage(stringToBytes32(_DomainName))  OnlyWinner(stringToBytes32(_DomainName)) ToBidAuctionAtStage(stringToBytes32(_DomainName),Stages.Ended) returns (bool)
    {
        var Domainname = stringToBytes32(_DomainName);
        //Return back the Winner's deposits.
        uint amount = AuctionLists[Domainname].pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the recipient
            // can call this function again as part of the receiving call
            // before `send` returns.
            AuctionLists[Domainname].deposits[msg.sender] = 0;

            if (!msg.sender.send(amount)) {
                // No need to call throw here, just reset the amount owing
                AuctionLists[Domainname].deposits[msg.sender] = amount;
                return false;
            }
        }

        //Transfer the highest bid to the Auction Owner.
        AuctionLists[Domainname].Owner.transfer(AuctionLists[Domainname].highestBid);
        //Changes the ownership of the DomainName and transfers it to the auction's Winner by calling the Set_Auction_Result from the Ghazal contract.
        Set_Auction_Result(Domainname);
        AuctionLists[Domainname].stage = Stages.Ended;
        AuctionLists[Domainname].AuctionisValue = false;
        return true;

    }
//
```

```
    ****************************************************************************************************

    function Transfer_Domain (string _DomainName, address _Reciever, bytes32 _TLSKey
        , string _IP_Address) public CheckAuctionStage(stringToBytes32(_DomainName))
         ToBidAuctionAtStage(stringToBytes32(_DomainName),Stages.Ended) OnlyOwner(
        stringToBytes32(_DomainName))
    {
        super.Transfer_Domain(_DomainName, _Reciever, _TLSKey, _IP_Address);
    }
//
    ****************************************************************************************************

}
```