

T.R.
GEBZE TECHNICAL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

**EFFICIENT PKI MODELS WITH CERTIFICATE
TRANSPARENCY BASED ON BLOCKCHAIN**

MURAT YASİN KUBİLAY
A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER ENGINEERING

GEBZE
2020

T.R.
GEBZE TECHNICAL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

**EFFICIENT PKI MODELS WITH CERTIFICATE
TRANSPARENCY BASED ON BLOCKCHAIN**

MURAT YASİN KUBİLAY
**A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER ENGINEERING**

THESIS SUPERVISOR
PROF. DR. HACI ALİ MANTAR

GEBZE
2020

T.C.
GEBZE TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BLOKZİNCİR TEMELLİ SERTİFİKA
ŞEFFAFLIĞINI SAĞLAYAN AÇIK
ANAHTAR ALTYAPISI MODELLERİ

MURAT YASİN KUBİLAY
DOKTORA TEZİ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

DANIŞMANI
PROF. DR. HACI ALİ MANTAR

GEBZE
2020



GTÜ Fen Bilimleri Enstitüsü Yönetim Kurulu'nun 08/05/2020 tarih ve 2020/24 sayılı kararıyla oluşturulan jüri tarafından 01/06/2020 tarihinde tez savunma sınavı yapılan Murat Yasin KUBİLAY'ın tez çalışması Bilgisayar Mühendisliği Anabilim Dalında DOKTORA tezi olarak kabul edilmiştir.

JÜRİ

ÜYE

(TEZ DANIŞMANI)

: Prof. Dr. Hacı Ali MANTAR

ÜYE : Prof. Dr. İbrahim SOĞUKPINAR

ÜYE : Prof. Dr. Haluk Rahmi TOPÇUOĞLU

ÜYE : Prof. Dr. Albert LEVİ

ÜYE : Dr. Öğr. Üyesi Zafeirakis ZAFEİRAKOPOULOS

ONAY

Gebze Teknik Üniversitesi Fen Bilimleri Enstitüsü Yönetim Kurulu'nun

...../...../..... tarih ve/..... sayılı kararı.

SUMMARY

In conventional PKI, CAs are assumed to be fully trusted. However, in practice, CAs' absolute responsibility for providing trustworthiness caused major security and privacy issues. To prevent such issues, Google introduced the concept of Certificate Transparency in 2013. Later, several new PKI models are proposed to reduce the level of trust to the CAs. However, all of these proposals are still vulnerable to split-world attacks if the adversary is capable of showing different views of the log to the targeted victims. In this thesis, we propose two new PKI architectures with certificate transparency based on blockchain, what we called CertLedger and KORGAN, to eliminate the split-world attacks and to provide certificate/revocation transparency.

In CertLedger, all TLS certificates' validation, storage, and entire revocation process is conducted as well as Trusted CA certificate management. It provides a unique, efficient, and trustworthy certificate validation process eliminating the conventional inadequate and incompatible certificate validation processes implemented by different software vendors.

In KORGAN, we eliminate the security and privacy issues of a recently proposed PKI architecture (Yakubov et al.'s scheme) and modify it in such a way that the validators (i.e., the consensus nodes) utilize a dynamic threshold signature scheme to generate signed blocks during the PBFT consensus mechanism. Hence, the trust to intermediary entities can be completely eliminated during certificate validation.

We implement both of our proposals on private Ethereum network to demonstrate the experimental results. The results show that our proposals have negligible overhead during TLS handshake and the certificate validation duration is significantly improved compared to the current TLS protocol.

Key Words: PKI, SSL/TLS, Certificate Transparency, Blockchain, Dynamic Threshold Cryptography, PBFT

ÖZET

Geleneksel Açık Anahtar Altyapısı (AAA) mimarisinde Sertifika Makamları (SM) bütünüyle güvenilir varlıklardır. Halbuki gerçek hayatta SM'lerinin güvenilirliği sağlamadaki mutlak sorumluluğu çok ciddi güvenlik ve mahremiyet sorunlarına yol açmıştır. Bu sorunları önlemek için, Google, 2013 yılında Sertifika Şeffaflığı konseptini önerdi. Daha sonra SM'larına duyulan güven seviyesini azaltmak için başka yeni açık log tabanlı sertifika modelleri de önerildi. Fakat bu modeller, saldırgan kurbanlara sertifika loglarını farklı şekilde gösterebilme imkanına sahip ise ayırık dünya saldırılarını engelleyememektedir. Bu tezde blokzincir temelli sertifika şeffaflığını sağlayan ve ayırık dünya saldırılarını engelleyen, adını CertLedger ve KORGAN olarak belirlediğimiz yeni iki adet AAA modeli öneriyoruz.

CertLedger'da tüm TLS sertifikaları doğrulanmakta, saklanmakta, tüm sertifika iptal ve güvenilir sertifika yönetim süreçleri gerçekleştirilmektedir. CertLedger tekil, verimli ve güvenilir bir sertifika doğrulama süreci oluşturarak günümüzdeki birbirinden farklı, yetersiz ve uyumsuz sertifika doğrulama yazılımlarına olan ihtiyacı da ortadan kaldırır.

KORGAN'da, kısa zaman önce Yakubov ve arkadaşları tarafından önerilen AAA modelinin güvenlik ve mahremiyet problemleri giderilip, PBFT konsensüs sürecinde doğrulayıcı düğümlerin yeni blokları dinamik eşik imza yöntemini kullanarak imzalaması sağlanmaktadır. Bu şekilde, sertifika doğrulama esnasında ara bileşenlere güvenilmek zorunda kalınmayacaktır.

Önerilerimizi, deneysel sonuçlarını gözlemlemek için Ethereum ağı üzerinde gerçekledik. Sonuçlar, TLS el sıkışması esnasında, önerilerimizin ihmal edilebilir bir ek maliyet getirdiğini ve sertifika doğrulama süresinin mevcut TLS protokolüne göre önemli ölçüde iyileştiğini göstermektedir.

Anahtar Kelimeler: AAA, SSL/TLS, Sertifika Şeffaflığı, Blokzincir, Dinamik Eşik Kriptografi, PBFT

ACKNOWLEDGEMENTS

First, I would like to thank my advisor Prof. Dr. Hacı Ali Mantar for his guidance and support in the completion of this thesis.

Secondly, I would like to thank my colleague Assoc. Prof. Dr. Mehmet Sabır Kiraz. I appreciate all his contributions of time, ideas and long months of fruitful discussions.

Besides, I would like to thank Prof. Dr. İbrahim Soğukpınar and Prof. Dr. Haluk Rahmi Topçuoğlu for their valuable comments during the progress of the thesis.

Finally, it is my pleasure to thank my spouse, Aslıhan Kubilay for her patience and endless support to me.

TABLE of CONTENTS

	<u>Page</u>
SUMMARY	v
ÖZET	vi
ACKNOWLEDGEMENTS	vii
TABLE of CONTENTS	viii
LIST of ABBREVIATIONS and ACRONYMS	xi
LIST of FIGURES	xiii
LIST of TABLES	xiv
1. INTRODUCTION	1
1.1. Background and Motivation	1
1.2. Ongoing Security Issues	3
1.3. Contributions	4
2. PRELIMINARIES AND RELATED WORK	8
2.1. Cryptographic Background	8
2.1.1. Digital Signature with Public Key Cryptography	8
2.1.2. Hash Functions	8
2.1.3. Digital Certificates	8
2.1.4. Shamir's Secret Sharing	9
2.1.5. Threshold Cryptosystems	9
2.2. Public Key Infrastructure Background	10
2.2.1. Certificate Authority	10
2.2.2. Registration Authority	13
2.2.3. Certificate Revocation	13
2.2.4. Certificate Validation Services	13
2.3. Blockchain Background	14
2.3.1. Merkle Trees	14
2.3.2. Blockchain Architecture	16

2.3.3.	Consensus Algorithms	17
2.4.	Related Work	19
2.4.1.	Sovereign Key (SK) Cryptography	19
2.4.2.	Certificate Transparency (CT)	19
2.4.3.	Certificate Issuance and Revocation Transparency (CIRT)	22
2.4.4.	Accountable Key Infrastructure (AKI)	22
2.4.5.	Attack Resilient PKI (ARPKI)	23
2.4.6.	NameCoin, CertCoin, and PB-PKI	24
2.4.7.	Distributed Transparent Key Infrastructure (DTKI)	25
2.4.8.	Blockchain-based Certificate and Revocation Transparency	26
2.4.9.	A Blockchain-Based PKI Management Framework	27
2.4.10.	CertChain: Public and Efficient Certificate Audit Based on Blockchain for TLS Connections	28
2.4.11.	IKP: Turning a PKI Around with Decentralized Automated Incentives	29
3.	CERTLEDGER	30
3.1.	HOW PUBLIC BLOCKCHAIN SOLVES PKI PROBLEMS?	30
3.1.1.	Blockchain Characteristics for PKI	32
3.2.	CertLedger Architecture	35
3.2.1.	Entities	38
3.3.	Trust and Threat Model	41
3.4.	PKI Functionalities of CertLedger	41
3.4.1.	Adding a New Trusted CA Certificate.	41
3.4.2.	Changing Trust Status of a CA Certificate.	45
3.4.3.	Adding a New TLS Certificate.	46
3.4.4.	Revocation of a TLS Certificate	46
3.4.5.	Transferring Token.	47
3.4.6.	Secure Communication Through TLS	49
3.4.7.	Fraud Management	54
3.5.	Security Analysis	57
3.6.	Performance of CertLedger	62
3.6.1.	Storage	62
3.6.2.	TLS Communication	64

3.6.3.	Scalability of the Blockchain	65
3.7.	Implementation And Experimental Results	65
3.8.	Comparison	66
3.8.1.	Log Proofs	67
3.8.2.	TLS Handshake Performance	67
3.8.3.	Independent Logs	70
3.8.4.	Log Availability	70
3.8.5.	Auditing and Monitoring	70
3.8.6.	Key/Certificate Store Management	71
3.8.7.	Privacy	71
4.	KORGAN: AN EFFICIENT PKI ARCHITECTURE BASED ON PBFT THROUGH DYNAMIC THRESHOLD SIGNATURES	73
4.1.	Security and Privacy Analysis of The Yakubov Et Al.'s Scheme	74
4.2.	Our Proposal: KORGAN	76
4.2.1.	Our Motivation: Why Dynamic Threshold Signatures Based Permissioned Blockchains?	77
4.2.2.	Our Approach: PBFT Consensus Mechanism with Dynamic Threshold Signatures	80
4.2.3.	The CA Smart Contract in KORGAN	83
4.2.4.	Certificate Validation of KORGAN in SSL/TLS	85
4.3.	Security Analysis	91
4.4.	Implementation and Experimental Results	92
5.	CONCLUSION AND FUTURE WORK	95
	REFERENCES	97
	BIOGRAPHY	104
	APPENDICES	105

LIST of ABBREVIATIONS and ACRONYMS

<u>Abbreviations</u>	<u>Explanations</u>
<u>and Acronyms</u>	
AAA	: Açık Anahtar Altyapısı
AEAD	: Authenticated Encryption with Associated Data
AKI	: Accountable Key Infrastructure
ARPKI	: Attack Resilient PKI
ARCert	: ARPKI certificate
CA	: Certificate Authority
CA/B	: CA/Browser
CIRT	: Certificate Issuance and Revocation Transparency
CL	: Certificate Log
CLM	: Certificate Log Maintainer
CP	: Certificate Policy
CPS	: Certificate Practice Statement
CRL	: Certificate Revocation List
CT	: Certificate Transparency
DCP	: Domain Certificate Policy
DNS	: Domain Name System
DTKI	: Distributed Transparent Key Infrastructure
ECC	: Elliptic Curve Cryptography
ETSI	: European Telecommunications Standards Institute
GB	: Giga Byte
GHz	: Giga Herz
HTTPS	: Hypertext Transfer Protocol Secure
IETF	Internet Engineering Task Force
IKP	: Instant Karma PKI
ILS	: Integrity Log Server
ILSO	: Integrity Log Server Operator
ISO	: International Organization for Standardization
IT	: Information Technology
ITU	International Telecommunication Union

JSSE	: Java Secure Socket Extension
KB	: Kilo Byte
KDC	: Key Distribution Center
MB	: Mega Byte
MITM	: Man In The Middle
MLM	: Mapping Log Maintainer
MTH	: Merkle Tree Hash
MHT	: Merkle Hash Tree
OCSP	: Online Certificate Status Protocol
OS	: Operating System
OWASP	: Open Web Application Security Project
P2P	: Peer to Peer
PBFT	: Practical Byzantine Fault Tolerance
PoS	: Proof of Stake
PoW	: Proof of Work
PKBoard	: Public Key of the Board Members
PKI	: Public Key Infrastructure
RA	: Registration Authority
RAM	: Random Access Memory
RP	: Reaction Policy
SCT	: Signed Certificate Timestamp
SHA	: Secure Hash Algorithm
SK	: Sovereign Key
SKBoard	: Private Key of the Board Members
S/MIME	: Secure/Multipurpose Internet Mail Extensions
SM	: Sertifika Makamı
SMT	: State Merkle Tree
STH	: Signed Tree Hash
TLS	: Transport Layer Security
TS	: TimeLine Server
TTP	: Trusted Third Party

LIST of FIGURES

<u>Figure No:</u>	<u>Page</u>
2.1: Conventional PKI Architecture.	12
2.2: Merkle Hash Tree.	15
2.3: Sample Blockchain.	16
2.4: TLS System in Certificate Transparency.	20
2.5: Checks and Balances Among AKI Entities.	23
3.1: Do You Need a BlockChain.	33
3.2: CertLedger Architecture.	37
3.3: Functionalities of CertLedger.	43
3.4: Generating a Transaction for Adding a Trusted CA Certificate through Board Members Algorithm.	44
3.5: Adding a Trusted CA Certificate Transaction through Trusted CAs State Object Algorithm.	44
3.6: Adding a TLS Certificate through DomainStateObject Algorithm.	45
3.7: Revoking a TLS Certificate through DomainStateObject Algorithm.	48
3.8: TLS Handshake Extensions for CertLedger.	49
4.1: TLS System of Yakubov et al.'s Scheme.	79
4.2: TLS System of KORGAN.	87
4.3: Verify Header Algorithm.	88
4.4: Validate Certificate Chain Algorithm.	89
4.5: Validate Certificate Algorithm.	90

LIST of TABLES

<u>Table No:</u>	<u>Page</u>
3.1: Security Comparison of Log Based Approaches to Certificate Management.	61
3.2: TLS Handshake Performance Results.	66
3.3: Table 3.3 Comparison of Log Based Approaches to Certificate Management.	68
4.1: Experimental Results of TLS Handshake in KORGAN.	92
4.2: Certificate Validation Durations Based on # of TLS Certificates in KORGAN.	93
4.3: Certificate Validation Durations Based on Trustchain Length in KORGAN.	94

1. INTRODUCTION

1.1. Background and Motivation

Public key cryptography evolved due to two main challenges in symmetric cryptography. The first one is the difficulty of key distribution in a secure fashion among the communicants. The communicants should previously somehow share the secret keys, or there must be a key distribution center (KDC) which is in charge of this process on behalf of the communicants. However this design brings a prominent weakness as Diffie, who is one of the discoverers of public key cryptography, mentioned in [1]. KDCs can be a source of single point of failure and can be compromised which could jeopardize the security of the crypto systems. Secondly, when cryptography is widespread used overtime, not only in the military, but also for commercial and private purposes, then there would be a need to sign digital documents, so that the integrity and the source (authenticity) of them are guaranteed.

Public key cryptosystems rely on one (private) key for signature and a different but related (public) key for verification. In the algorithms utilized in these cryptosystems, it is computationally infeasible to determine the signature key given only the knowledge of the cryptographic algorithm and the verification key. In order to sign a message, a message owner first calculates the hash value of his message and then signs it with his private key. Afterwards he sends both the message and the signature to the recipient. The recipient can verify the signature by decrypting it with the public key of the message owner and comparing the result with the hash of the message. However, at this point, the question of ‘How will the recipient receive and trust the public key of the message sender?’ arises. There are several proposals to solve this issue such as sending or announcing the public key to a group with whom the messages are exchanged and establishing a web of trust, or by binding it to its owner identity by public key certificates. There are a couple of implementation of public key certificates, but the most widely used public key certificate format is defined by ITU-T in X.509 [2]. Very briefly, an X.509 certificate is a digital document which comprises the name of the certificate owner (subject), public key, a validity period and the name and the signature of the certificate issuer.

X.509 certificates are widely used in today's internet especially in the TLS protocol [3] which is the most widely used cryptographic protocol for secure communications providing end-to-end security between client/server applications to prevent Man-In-The-Middle (MITM) kinds of attacks such as eavesdropping, tampering, or message forgery so that a malicious third party secretly cannot intercept, change or modify the communication traffic [4]. These certificates are issued, revoked, and managed under a set of policies, roles, and cryptographic methods which have been modelled under the so-called Public Key Infrastructure (PKI). In conventional PKI, CAs are assumed to be trusted organizations which verify the identity of subjects (e.g., domain names) and issue certificates to the domains. In particular, CAs are expected to operate according to some rules which are announced as Certificate Policy (CP) and Certificate Practice Statement (CPS) documents. In the current trust model, CAs have the absolute responsibility to issue correct certificates for the designated subject. However, CAs can still be compromised and fake but valid certificates can be issued due to inadequate security practices or non-compliance with the CP and the CPS. During the last decade, there have been the following serious incidents.

- The Stuxnet [5] malware is signed by the private keys of two compromised Taiwanese CAs which targets to control a specific industrial system likely in Iran, such as a gas pipeline or a power plant.
- Comodo CA, which has a big share in SSL/TLS market is hacked in March 2011 [6]. One of its Registration Authority (RA) is attacked to issue 9 fraudulent certificates where the attacker is traced back to Iran.
- A Dutch CA DigiNotar is pawned in July 2011 [7]. 531 fraudulent certificates are issued for valuable domains such as *.google.com, *.windowsupdate.com and *.mozilla.com. These certificates could easily be used to distribute malicious Windows updates or Firefox plugins without taking attention. At least 300.000 unique IPs are detected using Google services through these certificates, which 99% of the traffic is from Iran.
- Trustwave CA has sold a subCA certificate for one of its subordinates. This subCA has issued fraudulent TLS certificates which are used to introspect TLS traffic [8].

- A Turkish CA Turktrust has mistakenly issued subCA certificates instead of TLS certificates in December 2012 [9]. These certificates are used to generate TLS certificates for traffic introspection. Google identified the fraudulent Google certificate via Chrome.
- A subCA of the Chinese CA CNNIC, which is located in Egypt, issued fraudulent TLS certificates for traffic introspection in March 2015 [10]. Later on, it is identified that CNNIC is operated without documented CPS.
- Lenovo Superfish has deployed a local CA in its products in 2015 [11]. This CA is used to inject ads into the TLS protected web sites. Since the CA's private keys had been in the computer RAM, they would have been easily used to introspect traffic.
- Symantec issued unauthorized certificates for Google domains in September 2015 [12]. Later on Symantec claimed that these certificates were produced for test purposes.
- Symantec purchased Blue Coat in May 2016 [13]. Blue Coat has devices to snoop encrypted internet traffic. Blue Coat became a SubCA under Symantec. This unification increased the skepticism.

These fatal incidents lead to many researches to distribute the absolute trust on CAs to multiple authorities. To detect fake but valid TLS certificates, key pinning [14], [15], crowd sourcing [16]-[21] and pushing revocation information to browsers [22], [23] are the initial solutions which are partly implemented but failed due to scalability problems. There are also other recent PKI architecture proposals which are based on public logs and blockchain. For instance, Certificate Transparency (CT) [24] is one of those public log based PKI architectures proposed by Google and is mandatory in Chrome since May 2018.

1.2. Ongoing Security Issues

In the existing public log based proposals (described in Section 2), a strong adversary who has the ability to control the trusted entities (e.g., CA, Log Operator) can apply split-world attacks by providing different views of the logs to the targeted victims [25]. While some of these proposals cannot detect this attack, others propose

to use gossip protocols to identify it by ensuring a consistent view of the log for the TLS clients [26]-[29]. Still, this attack can only be identified if 1) there are sufficient numbers of gossiping TLS clients and servers, 2) at least some of them are able to view the genuine log and request the consistency proof from the log. Moreover, implementing gossiping clients in a vast amount of applications is not straightforward.

Secondly, certificate revocation and validation processes have major problems in today's PKI as also described in [30]. Namely, for the revocation process, certificate owners have to rely on CAs which have the full responsibility to revoke the certificates and provide proper revocation services. However, a compromised or malfunctioning CA may not behave as expected. Browsers would then accept revoked certificates since they rarely check whether the certificates are revoked [30]. More importantly, checking the revocation status of a certificate using OCSP also causes privacy concerns [30]. Moreover, incompatible and inadequate implementations of certificate validation and revocation behaviour within browsers are also error prone [30].

Another major security issue is the necessity of trusted key management in TLS clients. TLS clients trust CA certificates or some other trusted entities' public keys to make a successful TLS certificate validation. In case of a compromise, removal of root certificates/keys from all the clients' trusted key stores brings burden (e.g., due to IT policy restrictions, OS configuration, network communication) and causes vulnerability (e.g., outdated OS or apps). Moreover, if an adversary can inject a fake CA certificate to the trusted certificate store of a client, he can easily perform MITM attack without being detected [11], [31].

1.3. Contributions

In the first stage of our study, we propose a new and efficient PKI architecture, what we called CertLedger, which uses a blockchain based public log to validate, store, and revoke TLS certificates to make TLS certificates and their revocation status transparent while eliminating the above-mentioned issues. In summary, CertLedger provides the following features:

- Resistance to MITM kinds of attacks
- More transparent revocation process rather than relying on CAs

- Unique, efficient, and trustworthy certificate validation process
- Trusted key store elimination in the TLS clients
- Preserving privacy of TLS clients during TLS handshake
- No external auditing requirement due to inherent public log architecture
- Efficient and prompt monitoring to minimize the attack duration
- Transparency in trusted CA management

We provide a detailed security and performance analysis of CertLedger. Furthermore, in order to demonstrate the experimental results, we implement a prototype (except validating certificates due to constraints in parsing them) on private Ethereum network [32]. In the prototype, we modify a TLS client/server and introduce new TLS extensions to realize certificate validation, and experiment TLS Handshake with CertLedger. Finally, we evaluate the experimental outputs which basically shows that it is also feasible and quite efficient.

In the second stage of this study, we visit one of the most recent blockchain-based proposals for PKI (i.e., the Yakubov et al.'s scheme [63]), and address its security and privacy issues during TLS handshake. More concretely, TLS clients can easily be deceived by fraudulent full nodes or web services during certificate validation, because they cannot verify the validity of the incoming responses. Besides, fake but valid certificates also cannot be detected since only the hash values of the certificates are stored in the blockchain. Finally, a malfunctioning CA may not revoke a compromised certificate in a reasonable timeframe which would allow an attacker to exploit this vulnerability before changing the revocation status of the certificate. In order to eliminate these issues, we improve their scheme and propose a new PKI architecture. In summary, our scheme provides the following features.

- We use Practical Byzantine Fault Tolerance Algorithm (PBFT) [33] as the consensus mechanism where the consensus nodes hold a share of the blockchain signing key and a block can only be generated if a threshold number of them approve the block by signing it by their partial key share [35]. Since we use a dynamic threshold signature scheme, once the TLS clients receive the blockchain verification key (i.e., public key), they will not require to change it even if the set of validators is dynamic.

- During TLS handshake, TLS clients can validate the certificates without requiring to be a peer of the blockchain network. Moreover, they do not need to make any further network connection and query other entities during this process. In this respect, certificate and revocation transparency is now fully provided so that the TLS certificates and their revocation status are publicly monitored. Moreover, the privacy of the TLS clients is fully preserved during certificate validation.
- CAs are not the sole authority to revoke certificates anymore, certificate owners can now also revoke their certificates (if they still possess the private key). Therefore, the risk of not revoking a compromised certificate in a reasonable timeframe by a malfunctioning CA is minimized.
- We implement a prototype of KORGAN on Ethereum, and experiment certificate validation. Our experimental results show that TLS clients can validate certificates efficiently (in constant time) depending on only their processing power and memory. Moreover, TLS handshake overhead is insignificant in our scheme.

The rest of the thesis is organized as follows. In Section 2, we provide a brief cryptographic preliminary, PKI and blockchain background. We also shortly describe the recent public log based and blockchain based PKI architectures and point out their potential drawbacks. In Section 3,

- We describe how the afore-mentioned problems of PKI can be solved by using a public blockchain.
- We present the architecture of CertLedger, which is a new PKI model with certificate transparency based on blockchain.
- We provide a detailed security analysis of CertLedger and make a security comparison of it with the other proposals.
- We make a performance analysis of CertLedger in terms of storage requirements, TLS communication overhead and scalability.
- We discuss the prototype implementation of CertLedger on private Ethereum network [32] and evaluate its results.
- We compare CertLedger with the existing proposals in terms of performance.

In Section 4, we revisit one of the most recently proposed blockchain based PKI architectures (Yakubov et al.'s scheme [63]) and elaborate its security and privacy issues. Afterwards, we first describe our motivation to use dynamic threshold signatures based permissioned blockchains in our PKI architecture, and then describe PBFT consensus mechanism using dynamic threshold signatures. Then, we present our new PKI architecture, what we called KORGAN, which eliminates the highlighted issues and provides a more efficient construction than the existing schemes. Finally, we discuss our implementation and experimental results. In Section 5, we conclude the thesis and describe the future works.



2. PRELIMINARIES AND RELATED WORK

2.1. Cryptographic Background

2.1.1. Digital Signature with Public Key Cryptography

A public key digital signature scheme uses different keys for signature and verification. Private keys are only known by their owners and used for signature generation while public keys are publicly known by everyone and used for signature verification [36]. Only the owner of the private key can sign a message while the signed message can be verified by everyone by using the corresponding public key. Signature with a private key sk is denoted as $sig \leftarrow S_{sk}(msg)$, while verification of the signature by the corresponding public key pk is denoted as $msg \leftarrow V_{pk}(sig)$. Elliptic Curve Cryptography (ECC) [37], [38] and RSA [36] are the well-known digital signature schemes.

2.1.2. Hash Functions

A cryptographic hash function $H(m)$ maps an arbitrary size message m to a fixed size l -bit string $c \leftarrow H(m)$ [39]. Ideally, a random oracle model [40] is used to model a hash function which is a theoretical black-box responding to each unique request with a true random from its output domain. Since it records all the requests, it gives the same response for the same query whenever it is repeated. SHA-2 [41] and SHA-3 [42] are well known hash function families.

2.1.3. Digital Certificates

A public key of a user is completely useless, if the user can not convince the others that he is the sole owner of the corresponding private key. A public key certificate is a digital document which binds a certain public key to a certain entity. It should contain enough information for the relying entities to validate and verify the

identity of the owner of the certificate. Therefore, it should be signed by an authority to provide its integrity and authenticity.

ITU X.509 standard [2] is the most widely used certificate format in today's internet. In this format, a certificate is comprised of several fields such as subject (name of the owner), issuer (name of the issuing authority), public key, validity period, key usage and the signature of the authority.

2.1.4. Shamir's Secret Sharing

Secret sharing refers to methods of distributing a secret value among a group of participants in such a way that, each party have a share from the secret [43]. The individual shares of the secrets are useless and the secret can only be constructed if a sufficient number of the shares are combined. There are various proposed secret sharing schemes but we only describe Shamir's scheme as an example.

In Shamir's secret sharing scheme, proposed by Adi Shamir [43], a group of n users share a secret data D . In this scheme any predefined $(k + 1) \leq n$ or more users can reconstruct the secret. However, k or less users can not recover any information about D . This scheme can also be referred to as $(k + 1, n)$ -threshold secret sharing scheme, where $(k + 1)$ is the threshold and n is the number of users sharing the secret.

All users have a different point in two-dimensional plane, $(x_1, y_1), \dots, (x_n, y_n)$. All of the points must be chosen such that they are on a k -degree polynomial. Therefore, any $k + 1$ of these shares suffices for Lagrange's interpolation. The secret value is the evaluation of the polynomial on axis $x = 0$.

2.1.5. Threshold Cryptosystems

Only a single party owns the private key in basic public key cryptography. The main goal of a (t, l) -threshold cryptosystem is to distribute the shares of a private key among the parties $P_1, \dots, P_l$ in such a way that at least t of these holders are required for successful signing where $1 \leq t \leq l$. The private key remains secret all the times unless t or more of the parties decide to combine their key shares. A threshold cryptosystem has the following three components [44].

Distributed key generation. A protocol between $P_1, \dots, P_l$ for generating a public key h such that party P_1 obtains a private share x_i (of the private key x corresponding to h) and a public verification key h_i , $1 \leq i \leq l$. The protocol depends on t .

Threshold signature. A protocol between any set of t parties $P_{i1}, \dots, P_{it}$ that on input of a plaintext M , private shares $x_{i1}, \dots, x_{it}$, and verification keys $h_{i1}, \dots, h_{it}$ outputs signature S .

Verification. An algorithm that on input of a signature S , a public key h , outputs a plaintext M of S under public key h .

2.2. Public Key Infrastructure Background

Public key cryptography supports confidentiality, integrity, authentication, and non-repudiation security services. However, a carefully planned infrastructure is required to successfully implement these security mechanisms. A Public Key Infrastructure (PKI) is a combination of technology, processes, policies and technology needed to create, manage, use, store and revoke digital certificates in a public key cryptography scheme. A PKI implementation provides mechanisms to ensure establishment and maintenance of trusted relationships. It is used to enable security services for many applications such as TLS, IPsec, HTTPS and S/MIME.

A PKI framework is comprised of security and operational policies, security services, and interoperability protocols which supports management of keys and certificates using public-key cryptography. Certification Authorities (CAs) and Registration Authorities (RAs) are used to generate, distribute and manage the digital certificates. Directory services are generally used to publish encryption certificates. In a PKI, it is also possible to establish a trust hierarchy which is also used for providing risk management controls.

2.2.1. Certificate Authority

CAs establish an individual trust relationship with the entities which do not know each other. The CA authenticates the entity according to previously established criterias which are also documented as Certificate Practices Statement (CPS) and issues a digital certificate to the entity. CA confirms the identity of the certificate by

signing the certificate. Consequently, entities which are unknown to each other can establish a trust relationship, since they trust that CA have performed an appropriate identity authentication and the signature on the certificate is a confirmation of this fact. The level of trust to a CA depends on the established and used policies and procedures by the CA to verify the identity of an entity.

In conventional PKI (Figure 2.1), the certificates (consequently the public keys) of CA's must be distributed to all entities which trust to the CA's. A Root CA is a CA which is at the top of the trust hierarchy, and there is no superior CA which confirms its certificate. The Root CA's have self-signed certificates and these certificates must distributed using a secure distribution protocol.



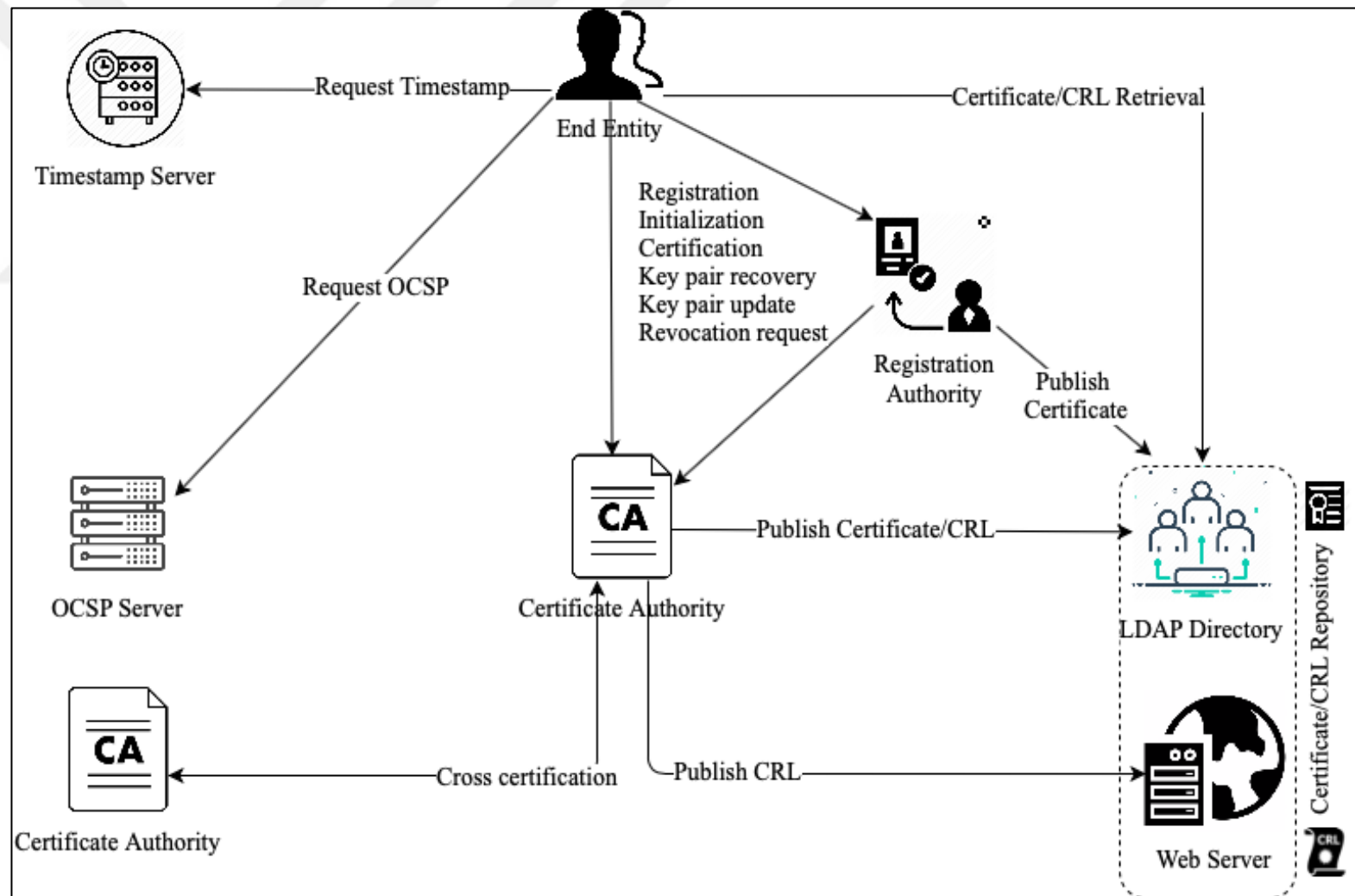


Figure 2.1: Conventional PKI Architecture.

2.2.2. Registration Authority

Registration Authorities are optional but common components of the PKI architectures. They perform administrative tasks on behalf of the CAs by their explicit permissions. The most important task of an RA is to verify the identity of the certificate requester and determine if he is entitled to receive or revoke a certificate. After a successful identity verification, it sends a certification or revocation request to the CA using certificate management protocols [45], [46] on behalf of the requester. The RA has to enforce all the policies defined in CPS while performing its operations.

2.2.3. Certificate Revocation

A certificate is valid for a certain period of time frame and expires after this period elapses. However, there are some situations when the certificate has to be revoked before the end of this validity period. For instance, the corresponding private key of the certificate can be compromised, lost or become unusable due to forgotten password. Moreover, if a change happens in the certified identity, or the owner of a certificate quits the company in a corporate environment, the related certificate can be revoked prematurely. Consequently, while verifying a cryptographic operation generated by the private key of a certificate, also the validity of the certificate has to be verified. For this purpose, the issuing CA (or a delegated authority) provides the validity status of the certificate by publishing Certificate Revocation Lists (CRL) or giving OCSP service.

2.2.4. Certificate Validation Services

CRL is a signed data structure which comprises the list of certificates which are considered no longer valid by their issuing CA. Briefly, in a CRL entry, the serial number of the certificate, its revocation time and revocation reason exists. A CRL has a validity date and a new CRL is supposed to be published before the previous one expires. When a certificate is validated by CRL, the status of the certificate is unclear between the revocation time of a certificate and the publishing time of the first CRL after revocation.

Online Certificate Status Protocol (OCSP), described in RFC 6960 [47], is used to check the revocation status of a certificate from its OCSP responder. Briefly, in this protocol, a list of certificate serial numbers, their CA name and key hashes are sent in the OCSP request. The corresponding response is signed and comprises an entry for each of the certificate in the request. The entry states if the certificate is valid, revoked or unknown. If it is revoked, then the revocation time is also included in the entry.

OCSP requires less bandwidth than CRL and the freshest revocation status of a certificate can be retrieved during certificate validation without waiting the publishing time of the next CRL if the OCSP responses are generated using real time status of the certificates.

2.3. Blockchain Background

2.3.1. Merkle Trees

Merkle Trees (Figure 2.2) are used to authenticate large amount of data [48]. A Merkle Tree is a binary tree where its' leaf nodes are generated from the hash values of a given list of input data. The output of the tree is a single hash value with the length of the used hash function's output value. The intermediate nodes of the tree are generated by hashing the concatenation of two child nodes. Consequently, the root node represents the fingerprint of all the input data and is named as Merkle Tree Hash (MTH).

The algorithm to calculate MTH for a given ordered list of n inputs, $D[n] = \{d(0), d(1), \dots, d(n-1)\}$ is described in [24] as follows:

Let $H(.)$ be a secure hash function, then the hash of an empty list is the hash of an empty string, so that

$$MTH(\{\}) = H(""). \quad (2.1)$$

The hash of a list with one entry (also known as a leaf hash) is:

$$MTH(\{d(0)\}) = H(0x00 || d(0)) \quad (2.2)$$

For $n > 1$, let k be the largest power of two smaller than n . MTH of an n -element list $D[n]$ is then defined recursively as

$$MTH(D[n]) = H(0x01 || MTH(D[0:k]) || MTH(D[k:n])), \quad (2.3)$$

where $||$ is concatenation and $D[k1:k2]$ denotes the list $d(k1), d(k1 + 1), \dots, d(k2 - 1)$ of length $(k2 - k1)$. The use of prefix 0 and 1 in leaves and nodes differs to give resistance to second preimage resistance.

The existence of a leaf node in a Merkle Tree can be verified using a set of additional nodes in the Merkle Tree which is called as Merkle Audit Path. The Merkle Audit Path for a leaf node comprises the shortest list of additional nodes to calculate the MTH of the tree. This path is also called as Merkle Audit Proof, since if MTH is calculated using this path, it can be used as a proof for the existence of the leaf node in the Merkle Tree.

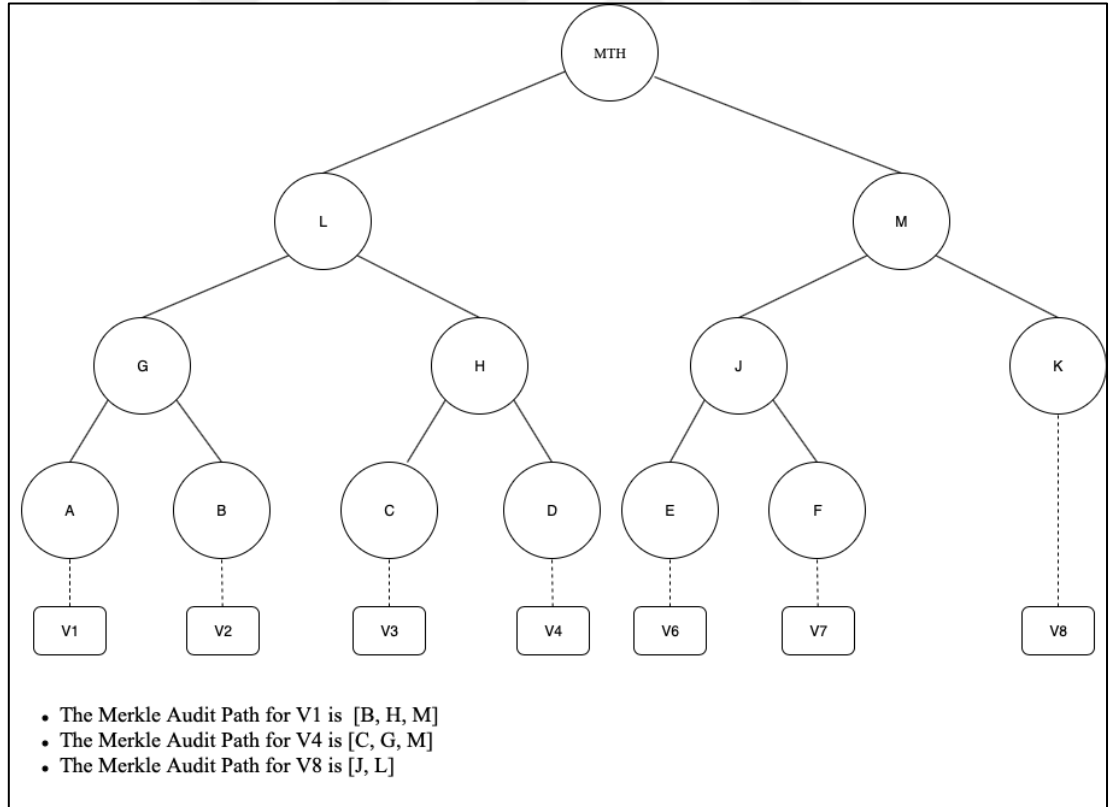


Figure 2.2: Merkle Hash Tree.

2.3.2. Blockchain Architecture

Blockchain is a shared, immutable, decentralized public ledger comprising an ever-growing list of blocks. A block is a data structure which consists of a header and a list of transactions. Each block is linked to the previous one with a cryptographic hash, therefore blocks are inherently secured from tampering and revision. A sample block, depicted in Figure 2.3, comprises the following fields.

- i) Block Number: The unique identifying number of the block.
- ii) Transaction Tree Hash: Merkle tree hash value of all the transactions in the block.
- iii) State Tree Hash: Merkle root hash of the state tree which holds the state of all accounts, smart contract codes and the data within the smart contracts
- iv) Timestamp: Current time as seconds with respect to a certain point of time (i.e., 1 January 1970 GMT).
- v) Nonce: It can have different purposes with respect to the consensus mechanisms. (i.e., Primarily used for increasing resistance against replay-attack in PBFT [33])
- vi) Previous block hash: n -bit hash value of the previous block

A Blockchain network is a decentralized peer-to-peer (P2P) network composed of full nodes and light nodes. Full nodes store a copy of the blockchain, validate, and propagate new transactions and blocks across the network while light nodes store only the block headers. All nodes can create transactions to change the state of the blockchain. New blocks of the transactions are collectively validated and appended to the existing chain according to a distributed consensus mechanism.

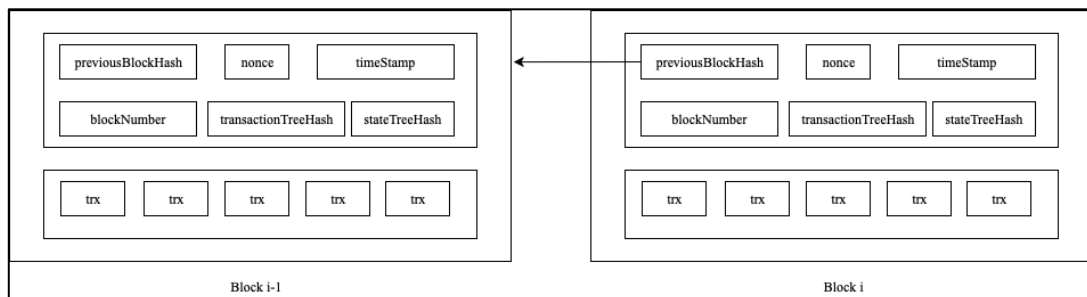


Figure 2.3: Sample Blockchain.

2.3.3. Consensus Algorithms

Reaching an agreement in a distributed environment is a challenge when some of the participants are faulty or misbehaving. Agreement upon the finality of the transactions in a blockchain network also has similar challenges and can be described with an analogy using famous Byzantine Generals Problem [50]. In this problem a group of generals surrender a city and have to agree on a battle plan. Generals communicate with each other through messengers, but the messengers can be captured by the enemy and not reach the destination. Moreover, some of the generals can be traitor and send false messages. The attack will not success if loyal generals act according to the plan and not adopt the plan due to a small number of traitors. Now, we describe some of the consensus protocols used within the todays blockchain networks.

2.3.3.1. PoW (Proof of work)

POW is a consensus mechanism used in Bitcoin network [51]. In PoW, a node who wants to create a new block first selects a list of transactions from a pool, validates them and generates the block. In order to generate the block, the node has to calculate a value (nonce) less than a certain value which is determined by the network. Calculation of the nonce requires certain amount of work which prevents randomly generated blocks. After the generation of the new block, which is also called as mining, the node (miner) dispatches it to the blockchain network. Other nodes in the network first validate the new block and then add them to their local blockchains. In Bitcoin network a new block is generated every 10 minutes, and the miner who has generated the new block is rewarded by a certain number of Bitcoins. It is also possible that more than one miner generates a new block at the same time and broadcast it to the blockchain network. In this case, some of the nodes add one of those new blocks to their local blockchains, while some of them add the others according to their retrieval time of the new blocks which will lead to temporary forks in the blockchain. However, the protocol ensures that, the chain with the maximum PoW (i.e., the longest chain) will survive while the others are discarded. However, if a node has the majority of the

mining power within the Bitcoin network, it can perform “51 %” attacks. In this attack, it can create a local branch by adding the new blocks it has generated which is longer than main branch. At some point this local branch converges as the main branch by discarding some of the previously added blocks. By this attack, it is possible to double spend the funds, and do not include some of the selected transactions into the new blocks.

2.3.3.2.PoS (Proof of Stake)

PoS [52] is initially discussed in Bitcoin forums as an alternative in order to avoid high energy demand in PoW. The general idea behind PoS is, the nodes in the blockchain has a certain amount of stake, and after a randomized leader selection, the selected node creates the new block. As in the PoW, the generator of the new node gets a reward, and receives the transactions fees. There are different stake approaches in PoS such as the amount or the age of cryptocurrency. However, PoS mechanisms faces mainly two main issues. The first issue is also called as *grinding attack* where malicious parties compromise the leader election process and collect more rewards than their fair share. Second issue is called as *nothing at stake* where construction of an alternative chain is much simpler than PoW, since there is no need to consume energy for solving the puzzle while generating a new block.

2.3.3.3.Practical Byzantine Fault Tolerance (PBFT)

PBFT algorithm, proposed by Castro and Liskov [33], aims to reach consensus through Byzantine nodes that tolerates Byzantine failures up to 1/3 malicious byzantine replicas. PBFT basically uses state-machine replication and replica voting for changing the state in the network. Nodes acting as validators in a PBFT network all have equal votes and the validation is run through multiple rounds to reach consensus. Signatures and encryptions are used between replicas and clients in order to reduce the complexity in terms of number of messages and make it scalable. A block is generated at the end of each round. A leader is selected in the beginning of each round according to some predefined rules. The leader is responsible for selecting the transactions from the transaction pool and creating the new block. Each round is

comprised of three phases which are called pre-prepared, prepared and commit. The consensus nodes enter to the next phase if they receive $2/3$ of the total votes. At the end of the three phases, all the honest nodes in the consensus group have the same view regarding to the state of blockchain by either accepting or rejecting the block proposal, thereby achieving the instant transaction finality.

2.4. Related Work

2.4.1. Sovereign Key (SK) Cryptography

SK has been proposed by Eckersley in order to prevent MITM and server impersonation attacks against domains protected by TLS certificates [53]. In SK, domains generate a sovereign asymmetric key pair for a set of selected services such as https, smtps, and imaps and publish public part of the key in a TimeLine Server (TS) along with their domain name. TS stores entries in read and append only data structures. During the TLS handshake, domains generate a fake certificate by their sovereign key and append to the certificate chain. If the clients cannot verify the signature on this fake certificate with the associated public key in TS, they refuse to make the connection to the service. However, TS does not return any verifiable proof to domains whether the sovereign key is appended to the log [54]. Furthermore, another strong assumption of SK is that clients have to trust not only the TS but also their Mirrors [54].

2.4.2. Certificate Transparency (CT)

CT aims to detect fake but valid certificates by providing append-only, publicly auditable logs for all issued TLS certificates, and shorten their lifetime [24]. CT uses append only Merkle Hash Trees (MHT) for appending and efficiently verifying TLS certificates [55]. In this respect, CT introduces Certificate Logs (CL), Monitors, and Auditors as new entities. CAs submit the new TLS (pre)certificates to several CLs and each of them generate a cryptographic proof called Signed Certificate Timestamp (SCT). After appending the new certificates to the log, CL computes the Signed Tree

Head (STH) and generates a Consistency Proof for proving whether the new log is an extension of the old one. Merkle Audit Proof shows the existence of a certificate in the log. SCT can be delivered to the TLS clients either as a certificate extension, or in a TLS extension during TLS handshake, or in the OCSP response. If the TLS clients do not receive SCT by aforementioned-means or cannot verify them using CL's public key, they may refuse to connect to the service.

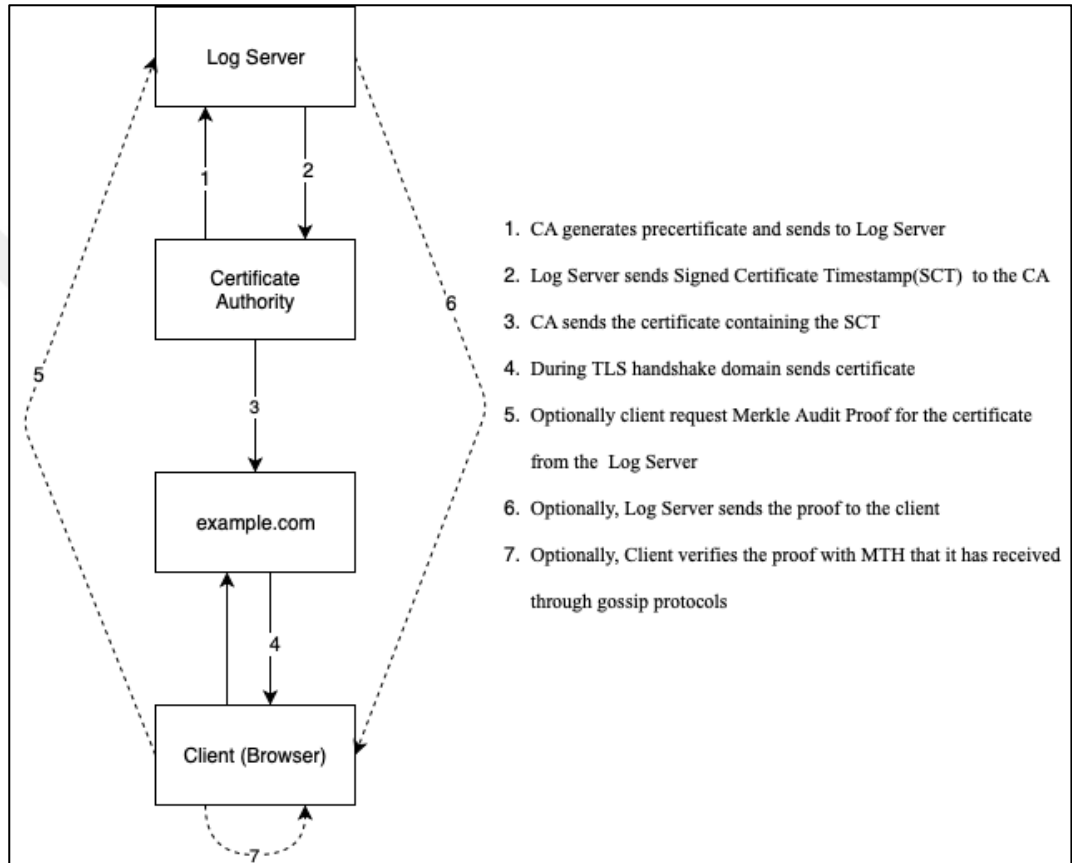


Figure 2.4: TLS System in Certificate Transparency.

Domain owners or private companies can act as Monitors and continually inspect certificates of their interest in CL whether there exist illegitimate certificates. Browsers, or in general TLS clients, act as Auditors. They verify if a CL is behaving properly and cryptographically consistent.

However, the authors in [25] show that if an adversary can get a fake but valid TLS certificate and can control the CL then he can perform an impersonation/MITM attack to the targeted victims by providing a fraudulent view of the log that contains the fake certificate. This attack is later referred to as *split-world attack* [26] which is elaborated as follows.

- i) The adversary gets a fake but valid TLS certificate for a domain from the CA.
- ii) The adversary or the CA submits this certificate to the log which is appended to the fraudulent hash tree.
- iii) The adversary obtains a bogus SCT for the fake TLS certificate from the log. SCT is fake, since the log maintains more than one hash trees. It shows different views of the tree to different sets of clients. Merkle Audit Proof and Consistency Proof are generated from different hash trees and the victim clients cannot understand that the proofs are not generated from the genuine hash tree.
- iv) When a targeted victim client tries to connect to a domain, the adversary controlling the traffic sends the fake certificate and the bogus SCT to the client.
- v) Upon validation of the fake certificate and the SCT, the client connects to either the fake domain or the real domain through a proxy.

Monitors cannot detect this attack since they get and verify the Consistency Proof's only from the genuine view of the log. Auditors (victim TLS Clients) cannot detect the attack, since they can verify the existence of the fake TLS certificate in the log with the fake STH and the Merkle Audit Proof generated from the fraudulent view of the log. The other Auditors also cannot detect either, since they get STH and the proofs from only the genuine view of the log.

As pointed out in RFC 6962 for CT [24], to detect the attack in the existing CT architecture, there must be sufficiently large number of clients and servers gossiping their view of STHs with each other. Moreover, some of the clients should also act as Auditors for checking the consistency of the log. Some of these gossiping Auditors should be able to receive both the genuine and the fake proofs belonging to a log to figure out the inconsistency. If an Auditor has two STHs with the same tree size but with different values, this will be an evidence for the misbehavior of the log. If an Auditor has two different STHs with unequal tree sizes then it can request the Consistency Proof from the log. Since the log cannot provide this proof it should be investigated for malicious behavior.

2.4.3. Certificate Issuance and Revocation Transparency (CIRT)

CIRT focuses on extending CT to provide additional proofs so that CAs can show their honest behavior [56]. Therefore, CIRT maintains two Merkle trees as public logs which are ordered chronologically and lexicographically. In addition to CT's proofs, CIRT provides proofs for whether a certificate is marked as revoked in the log, whether a certificate is current (not replaced and not revoked) and finally whether a certificate is absent (has never been issued). However, CIRT is also vulnerable to the split-world attacks as CT.

2.4.4. Accountable Key Infrastructure (AKI)

AKI proposes a new PKI architecture for reducing the level of trust to the CAs where all the operations defined in a conventional PKI are performed with participation of more than one entity [57]. All entities either monitor or report the operations performed by other entities in order to distribute the accountability among the participating entities (see Figure 2.5). Certification Agency (CA), Integrity Log Server (ILS) Operators (ILSO), and Validators are the new entities introduced by AKI. CAs still issue certificates to domains but they are not the absolute authority for certificate management anymore. ILSOs store not only the certificates, but also their registration, update, and revocation information lexicographically according to domain names. Validators monitor ILS operations and check whether they operate as expected. Domain owners can select the CAs and the ILSOs they trust, define the minimum number of recommended CA signatures on a certificate and the rules for certificate management. They register the certificate to one or more ILSs. Domains send their certificates and the verification information received from the ILSs to the clients. Clients verify the certificates using preinstalled trusted CA certificates and the ILS public keys.

AKI makes a strong assumption that the trusted entities (CAs, ILSs, and Validators) do not collude with each other which is unlikely in case of a strong adversary who is willing to intercept the traffic. A compromised CA and an ILS is sufficient to generate a fake certificate in AKI. The adversary gets proofs from the compromised entities in order to send to the TLS client. Taking this fact into

consideration, it is also possible to make a split-world attack to the AKI. Unfortunately, there is no way of detecting this attack in AKI. Secondly, certificate revocation is a weak point in AKI since the domain owner can request the certificate revocation from an ILS without requiring confirmation of any other parties. Namely, an adversary, which has compromised the domain private key, can request the revocation of the corresponding certificate without further verification. Finally, clients have to trust not only the CAs and the ILSs but also the Validators in AKI which is a burden for them in terms of trusted entity management.

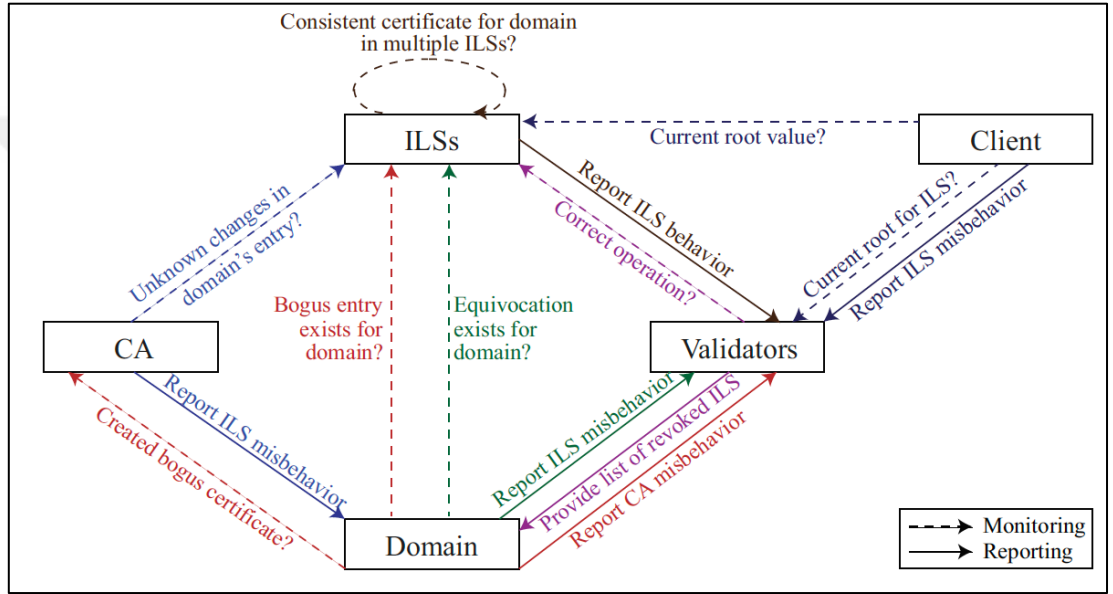


Figure 2.5: Checks and Balances Among AKI Entities [58].

2.4.5. Attack Resilient PKI (ARPKI)

ARPKI [58] is an improvement of AKI, which offers a security guarantee against adversaries which can compromise even $n - 1$ trusted entities. For generating an ARPKI certificate (called ARCert), at least two CAs and one ILS are required. For the initial registration process, a domain owner selects the trusted entities, at minimum two CAs and one ILS (i.e., CA_1 , CA_2 , and ILS_1). The domain owner designates one of the CAs (e.g., CA_1) for validating CA_2 and ILS_1 operations and serving as a messenger between these entities and himself. ILS_1 takes the responsibility for ensuring synchronization of ARCert among majority of ILSs. Domains send the cryptographic

proof signed by three trusted entities along with the ARCert to the clients. Upon verification of the proofs, clients connect to the domain.

We highlight that ARPKI is also subject to split-world attack if the entities (which are required to generate an ARPKI certificate) collude together. We also note that there is also no detection mechanism for this attack in ARPKI as AKI. Finally, designating an ILS for making synchronization with other ILSs may lead to a single point of failure in ARPKI.

2.4.6. NameCoin, CertCoin, and PB-PKI

NameCoin [59], which is a fork of Bitcoin [51], is designed to act as a decentralized DNS for “bit” addresses. In NameCoin, a self-signed TLS certificate of a domain can be added to DNS addresses as auxiliary information. TLS clients can then authenticate the domain during TLS handshake using this certificate.

CertCoin [60] proposes a decentralized PKI architecture, based on NameCoin, which eliminates the use of CAs. In Certcoin, the basic PKI operations are defined as registering an identity with a public key, and looking up, verifying, and revoking the public key for the given identity on the blockchain. In particular, identities register an online and an offline key pair where the online key is used for domain authentication whereas the offline key is used to revoke old online keys and to sign new online keys. However, in both proposals (i.e., Namecoin and Certcoin), there is no identity verification. Namely, whoever first claims the ownership of an identity owns it. In this way, in the real world, identities (in particular TLS clients) can easily be deceived. Secondly, they have also no adequate solution in case both online and offline keys are compromised. Therefore, the identity owners cannot reclaim their identities securely which can lead to unusable identities. Thirdly, since both proposals are using the Bitcoin blockchain architecture, verifying the owner of a public key, and looking up the public key of an identity is extremely inefficient. In order to solve these issues, Certcoin proposes extra entities which are also maintained in the blockchain such as accumulators and distributed hash tables. Since these entities are not a part of the blockchain architecture, they can cause new complexities in terms of maintenance, authentication, and verification.

PB-PKI [61] adapts CertCoin to be privacy aware by not linking the identity with the public key. Namely, it aims to avoid tracking the identities' actions by their use of public keys (which can be desirable for several use cases such as ubiquitous computing and the IoT, vehicular networks.). Although the proposal is interesting in such scenarios, it is unfortunately not suitable for identity management of domains as well as their certificate management lifecycle due to the contradicting the transparency requirement.

2.4.7. Distributed Transparent Key Infrastructure (DTKI)

DTKI [54] proposes a public log based certificate management architecture which minimizes oligopoly, prevents use of fake certificates, and claims being secure even if all service providers collude all together. Certificate Log Maintainer (CLM) and Mapping Log Maintainer (MLM) are the two new entities introduced by DTKI. CLMs keep all valid, revoked and expired certificates for a set of domains and provide proofs for existence or absence of them. MLM maintains the association between a set of domain names and the CLMs which are maintaining the logs for them. Mirrors maintain a full copy of the data stored by both the CLMs and the MLM. CAs make identity checks and issue certificates, but they are not the sole entity for providing trust to connect to a domain. Inspiring "sovereign key" concept in SK, a domain owns two types of certificates, TLS certificate and a master certificate which is used for requesting a new TLS certificate from the CA, and registering it to the CLM. Users or in particular browsers first query the MLM in order to find the correct CLM for a specific domain. To make a connection decision, first the proofs received from the MLM is verified, then CLM is queried in order to retrieve proofs for the domain's TLS certificate.

DTKI assumes all master certificates are genuine, and fake master certificate issuance does not likely occur since CAs are running businesses which cannot afford loss of reputation. However, this is not a valid argument since most of the fake certificates are generated due to lack of adequate security controls or processes. Namely, if the CA and the CLM are both compromised, DTKI would not be able to prevent fake Master and TLS certificate issuance. From this perspective, the adversary who is controlling CLM and capable of generating fake but valid Master and TLS

certificates can make split-world attack to the targeted victims. This attack unfortunately cannot be detected because there is no monitoring process in DTKI due to the assumption of genuine master certificates.

2.4.8. Blockchain-based Certificate and Revocation Transparency

Very recently in FC'18, Wang et al. proposed a blockchain based certificate and revocation transparency to store the TLS certificates and their revocation status (i.e., CRL and OCSP) [62]. Briefly, in this scheme, web servers publish their TLS certificates to the blockchain using their publishing key pairs which are used to sign the transactions. The publishing key pairs are different from the key pair in the certificate and are initially certified by a certain set of web servers which already exist in the blockchain. In this scheme, the transactions have a validity period, therefore TLS certificates and their revocation status are added to the blockchain periodically during their lifetime. During a TLS handshake, a web server sends a certificate transaction and its Merkle audit path to a TLS client which verifies its validity through its locally stored synchronized block headers. This proposal has the following drawbacks.

- i) It has an unreliable basis for providing the trustworthiness of publishing key pairs. Namely, a strong adversary, who can get fake but valid TLS certificates from corrupted CAs, can create some bogus domains (i.e., web servers) in advance and can use them to generate a valid signature of a publishing key pair transaction. This problem occurs due to the trust to the web servers. The authors propose to solve this issue by having more publicly-trusted CAs to invalidate the interfering transactions. However, this introduces a trust level issue which is not explicitly clarified.
- ii) For the revocation transparency, it relies on the CAs to publish the revocation data of the TLS certificates on the blockchain. However, the compromised or malfunctioning CAs may not issue CRL or give OCSP response to the client in the specified time.
- iii) It is subject to MITM attacks where an adversary can convince a client with an unexpired transaction of a revoked TLS certificate. More concretely, during

a TLS handshake, web servers send a certificate transaction to a TLS client to validate the TLS certificate. The TLS client accepts this transaction if it is not expired and is added to a confirmed block. However, a revoked or updated TLS certificate can also have an unexpired certificate transaction in the blockchain. Therefore, once an adversary sends this unexpired certificate transaction with its Merkle proofs to a TLS client, it is accepted during the TLS handshake. The TLS clients cannot detect the final state of the certificate since the clients only check the existence of the transaction in the corresponding block.

iv) The proposal is also inefficient in terms of storage costs due to following design considerations.

- A TLS certificate is added to the blockchain periodically during its lifetime,
- A CRL can be added to the blockchain for each revoked certificate (i.e., the number of CRL insertion to the blockchain is equal to the number of revoked certificates),
- Publishing key pairs are added to the blockchain periodically
- It has large size headers which comprise DNS names existing in the transactions of the block.

2.4.9. A Blockchain-Based PKI Management Framework

In [63] Yakubov et al. proposed a blockchain-based PKI management framework for issuing, validating, and revoking X.509 certificates. For each CA, a smart contract is generated in the blockchain which includes the CA certificate, the hashes of the CA's issued certificates and their revocation status. They utilize new extensions to standard X.509 certificate to embed blockchain based meta data. One of these extensions is populated by the issuing CA's smart contract address, if the certificate is not a root CA certificate. This extension is used to construct a trust chain while validating a TLS certificate.

We point out that there is a trust issue with the root CA certificates in the proposal. Namely, an adversary can easily deploy smart contracts to the blockchain for fake root CA certificates, subCA certificates, and as well as TLS certificates since

there is no prevention mechanism. In this way, the adversary can apply a MITM kind of attack. Secondly, for certificate validation, TLS clients have to trust the answers coming from the web services or the full nodes which execute the certificate validation function of the smart contract. Namely, TLS clients cannot verify the validity of the certificate. Thirdly, it is impossible for domain owners to monitor fake TLS certificates in the blockchain, since only the hash value of the end entity certificates is stored in the blockchain. This is also rather critical since a detection capability is one of the most crucial requirements for transparency. Finally, only the CAs can revoke a certificate in the proposal. However, if the CA is compromised, then an adversary can perform MITM kind of attack.

2.4.10. CertChain: Public and Efficient Certificate Audit Based on Blockchain for TLS Connections

CertChain [64] proposes a blockchain-based public auditing architecture for TLS connections. It introduces a new data structure for blockchain transactions, called CertOper, which consists of certificate related data and certificate operations such as registration, update, and revocation. Furthermore, it also proposes a method which uses a dual counting bloom filter to eliminate false positives during revocation checking. The underlying consensus protocol of CertChain is based on the Ouroboros [52], which uses the dependability-rank of a CA or a bookkeeper for a leader selection.

However, this proposal is subject to a MITM kind of attack in case of dishonest bookkeepers. Assume that a bookkeeper behaves honestly while maintaining the blockchain and has a good dependability-rank but sends incorrect responses to the TLS clients. In this case, while validating a TLS certificate, a client launches a verification request to a bookkeeper which can send an incorrect response. Since the client is going to accept it as valid it will be subject to a MITM attack. Note that CertChain does not provide any verification mechanism for the response. Also, privacy of the TLS clients is not preserved since the bookkeepers can track the clients' visiting history.

2.4.11. IKP: Turning a PKI Around with Decentralized Automated Incentives

The authors in [65] propose a framework, called Instant Karma PKI (IKP), which aims to solve the following two specific problems: 1) insufficient incentives for CAs to invest more in security, and 2) time- and labor-intensive processes for reporting unauthorized certificates. In this framework, a domain can specify criterias (called Domain Certificate Policy (DCP)) where its TLS certificate must meet and CAs can sell insurance policies (called Reaction Policies (RP)) to domains. RPs are automatically running (via smart contracts) financial transactions when an unauthorized certificate is reported (i.e., as financial penalties). The framework introduces a new entity called IKP Authority which maintains information on CAs, DCPs, and RPs, and acts as a trusted mediator to ensure a fair exchange.

We note that IKP does not consider untrusted CAs, instead tries to incentivize them with a positive expected return on investment (to invest in higher security). They only consider strong adversaries that can access the long-term keys of the trusted CAs (and issue fake but valid certificates) (see [65, Section II-B]). Therefore, unlike CertLedger, IKP only proposes an incentive mechanism within the existing PKI architecture but does not propose a solution for the security issues that are highlighted in Section 1.1.

3. CERTLEDGER

3.1. HOW PUBLIC BLOCKCHAIN SOLVES PKI PROBLEMS?

Blockchain is a shared, immutable, decentralized public ledger comprising an ever-growing list of blocks. A block is a data structure which consists of a header and a list of transactions. Each transaction is generally formed as

$$trx := (M, Signature) \quad (3.1)$$

$$M := (PKSender; receiver; data) \quad (3.2)$$

$$Signature := (SignSKSender(H(M))) \quad (3.3)$$

where *PKSender* denotes the public key of a sender (address of the sender is derived from *PKSender*), *receiver* is the address of a receiver, *SKSender* is the private key of the sender, and $H(.)$ is a secure hash function. Each block is linked to the previous one with a cryptographic hash, therefore blocks are inherently secured from tampering and revision.

A Blockchain network is a decentralized peer-to-peer (P2P) network composed of full nodes and light nodes. Full nodes store a copy of the blockchain, validate, and propagate new transactions and blocks across the network while light nodes store only the block headers. All nodes can create transactions to change the state of the blockchain. New blocks of the transactions are collectively validated and appended to the existing chain according to a distributed consensus mechanism.

There are several consensus mechanisms used in blockchain networks, e.g., Practical Byzantine Fault Tolerance algorithm (Practical BFT) [33] (which is utilized in HyperLedger Fabric [66]), Delegated BFT [67] (which is utilized in Neo [68], and an improved version VBFT is utilized in Ontology [69]), Proof-of-Work (PoW) (which is utilized in Bitcoin [51], Ethereum [32]), Ripple [70] (which is utilized in Ripple [71]), and Proof-of-Stake (PoS) [52] (which is utilized in Cardano [72],

PeerCoin [73]). From now on, for simplicity, we refer to “blockchain” as blockchain network. In general, there are also three types of Blockchain:

- i) Permissionless (e.g., Bitcoin [51], Ethereum [32], ZCash [74]),
- ii) Public Permissioned Blockchain (e.g., Ripple [70]),
- iii) Private Permissioned Blockchain.

While any peer can join and leave the network any time in permissionless blockchains, permissioned blockchains require authorization for the membership of the peers. In the permissioned blockchains, if public verifiability is required, then the transactions of the blockchain must also be public. However, private permissioned blockchains may be used for corporate networks which have sensitive data. In the following, we describe why blockchain solves

- i) Split-world attack,
- ii) Certificate revocation and validation problems, and
- iii) Trusted certificate/key store management problems.

First, as described in Section 2.4, existing proposals try to solve the transparency issue in PKI by introducing (one or more) public logs [24], [55], [58], [59]. However, they are still subject to split-world attack due to the trust to a log operator. In order to prevent such an attack, the trust should be distributed in such a way that a single log operator could not be able to control the log itself. Therefore, a decentralized public log mechanism is required which is synchronously updated only upon a consensus of its clients. Once the consensus is achieved, the log should be updated and could not be reverted anymore. We would also like to highlight that, in order to prevent the split-world attack, RFC 6962 for CT states [24],

“All clients should gossip with each other, exchanging STHs at least; this is all that is required to ensure that they all have a consistent view. The exact mechanism for gossip will be described in a separate document, but it is expected there will be a variety”.

Therefore, as also directly implied by “all clients gossiping to each other”, blockchain is an architecture fulfilling all these required features.

Second, certificate validation is a required process for each TLS connection, and comprises trusted path construction and revocation checking phases. Currently, certificate validation burden is entirely on TLS clients (e.g., internet browsers). However, if TLS certificates are stored on a blockchain, trusted path construction can be done only once while they are being appended to the blockchain. Consequently, browsers can trust the TLS certificates on the blockchain without requiring further validation once they obtain the required Merkle proofs. As also described in [30], browsers have different implementations for revocation checking which are error prone. By managing the revocation status of TLS certificates on the blockchain, revocation checking process can be simplified and unified. Namely, revocation checking burden using CRL and OCSP services can be eliminated which will also preserve the privacy of the TLS clients [30]. Furthermore, while certificates can be revoked only by CAs in the existing system, they can also be revoked by their owners on the blockchain. When the revocation process is conducted through the blockchain it becomes more transparent.

Finally, validation of certificates while appending to the blockchain also requires trusted CA certificates to exist on the blockchain. Therefore, these certificates have to be stored and managed on the blockchain as well. Hence, TLS clients do not need to store trusted CA certificates anymore since the blockchain ensures to append the TLS certificates issued only from the trusted CAs.

3.1.1. Blockchain Characteristics for PKI

Using the decision-sequence (Figure 3.1) of Wust and Gervais [100], we below identify the type of blockchain to manage the certificate log. Note that a *writer* is an entity who is able to accumulate new transactions into a new block and append it to the blockchain.

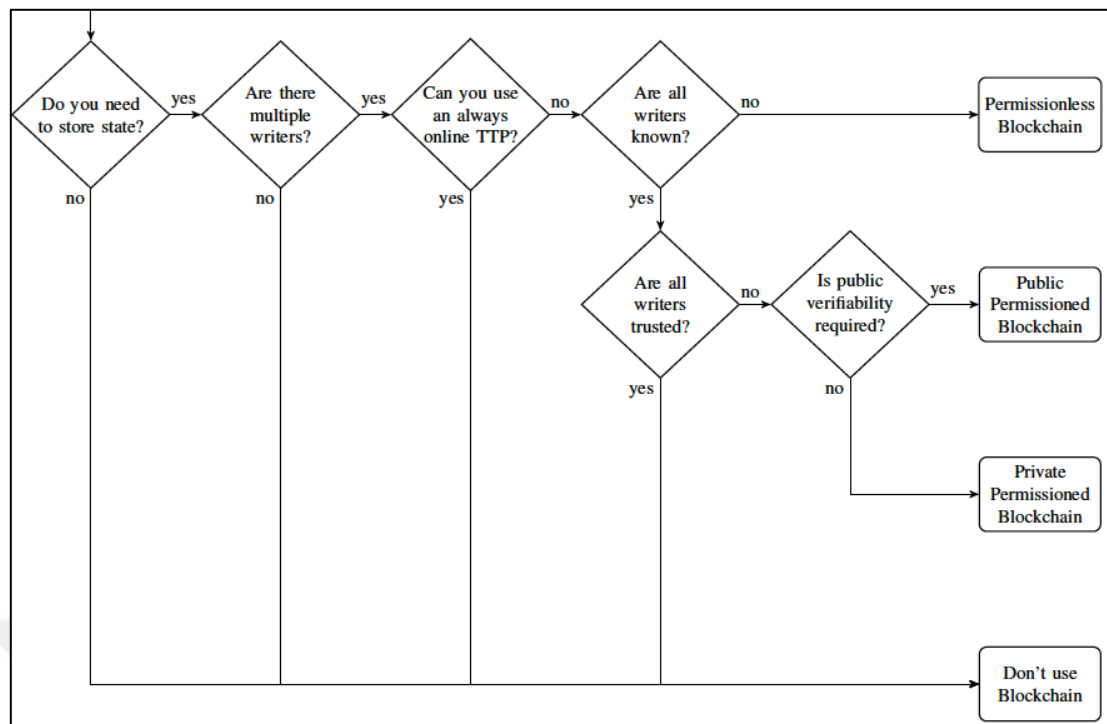


Figure 3.1: Do You Need a Blockchain [100].

- i) Do you need to store state? TLS certificates are being updated continually due to expiration or revocation. The state of the certificates has to be stored and updated whenever necessary.
- ii) Are there multiple writers? TLS certificates generated by the trusted CAs are appended to the log. A manipulated single writer can append fake but valid certificates to the log, delay or ignore appending the genuine ones. Therefore, increasing decentralization while writing to the log will reduce the risk of manipulation due to the fact that broad participation of writers will lead to a more reliable and robust log.
- iii) Can you use an always online Trusted Third Party (TTP)? As described previously, a strong adversary is assumed to control a TTP which may lead to a single point of failure. Therefore, we cannot use an online TTP as it is the main source of vulnerability.
- iv) Are all writers known? The writers may be known or unknown. However, if they are known, they should be selected and dispersed all over the world in such a way that their malicious cooperation and manipulation could not be possible.

- v) Are all writers trusted? Even though, all the writers are seemingly trusted, some of them can be controlled by a strong adversary.
- vi) Is public verifiability required? The existence and validity status of all TLS certificates must be verifiable by public for ultimate transparency.

Thus, the decision flowchart results in either a permissionless or a public permissioned blockchain for managing the certificate logs. However, the following five additional features are required in the underlying blockchain.

- It must comprise smart contract infrastructure to implement the required rules for validating the state transitions [75].
- The underlying consensus mechanism should avoid possibility of forks, since some of the TLS clients can verify an incorrect state of a TLS certificate before the blocks have been fully confirmed.
- The required time for the confirmation of a new block in the consensus mechanism should not be high, so that the transactions can change the state of the TLS certificates in an acceptable time frame.
- Blockchain architecture should enable TLS clients to verify the final state of the TLS certificates efficiently¹.
- TLS clients should be able to identify whether a block (or only its header) is genuine or not.

Since we expect that most of the TLS clients will be light nodes of the blockchain, and it should not be very expensive for them to verify the genuineness of a block header. In this respect, any consensus mechanism, which does not lead to temporary forks, can be used in the underlying blockchain architecture of CertLedger (e.g., PBFT [33], DBFT/VBFT [67], [69])². Neo [68], Ontology [69], and EOS [76] are some of the candidate blockchain architectures fulfilling these requirements. For

¹ State Merkle Patricia Trees are generally maintained to generate proofs and track the final states of the assets efficiently. The Merkle root of this tree is stored in the block headers, so that the integrity of the tree, and the state proofs generated out of it can be verified [104].

² To the best of our knowledge, the above-mentioned requirements are currently fulfilled by only permissioned blockchains.

more efficiency, their underlying consensus algorithm can be modified in such a way that a block can only be confirmed only upon a sufficient number of nodes sign it via a threshold signature mechanism, e.g., $(n + 1; 3n)$ [77], [78] (a consensus outcome (i.e., new block) can now be verified by anyone without requiring an additional consensus algorithm).

3.2. CertLedger Architecture

CertLedger is a PKI architecture (see Figure 3.2) to validate, store, and revoke TLS certificates and manage Trusted CA certificates on a public blockchain. It aims to make certificate issuance and revocation life cycle more transparent and to eliminate any kinds of MITM attacks. Moreover, it also aims to unify certificate validation process for all TLS clients due to its inconsistent and inadequate implementations throughout different TLS clients. More concretely, CertLedger manages the PKI functionalities through state objects. A state object is a digital document which is comprised of data and an immutable smart contract code to manage it. Each state object has a unique address in the blockchain. State changes of the assets are triggered by transactions and tracked through the state objects. CertLedger comprises the following state objects:

- **Trusted CAs State Object:** stores the set of the trusted CA certificates and smart contract code for adding new CA certificates as “trusted” or changing the status of the existing ones as “untrusted”. The smart contract code makes all the necessary controls on the certificate and checks whether it is compliant to international standard certificate profiles (e.g., RFC 5280 [79]) for being a CA certificate.
- **Domain State Object:** stores and manages the states of all TLS certificates and their revocation status. This state object comprises the necessary code for validating the TLS certificate according to international standards such as RFC 5280 [79]. It calls the Trusted CAs State Object while building a trusted path for the TLS certificate. Moreover, it also comprises the necessary code for changing the status of the TLS certificate as “revoked”.

- An Account State Object: stores the CertLedger token balance of a CertLedger entity and is controlled by the account's private key. This state object is used to create and trigger any transaction within CertLedger.



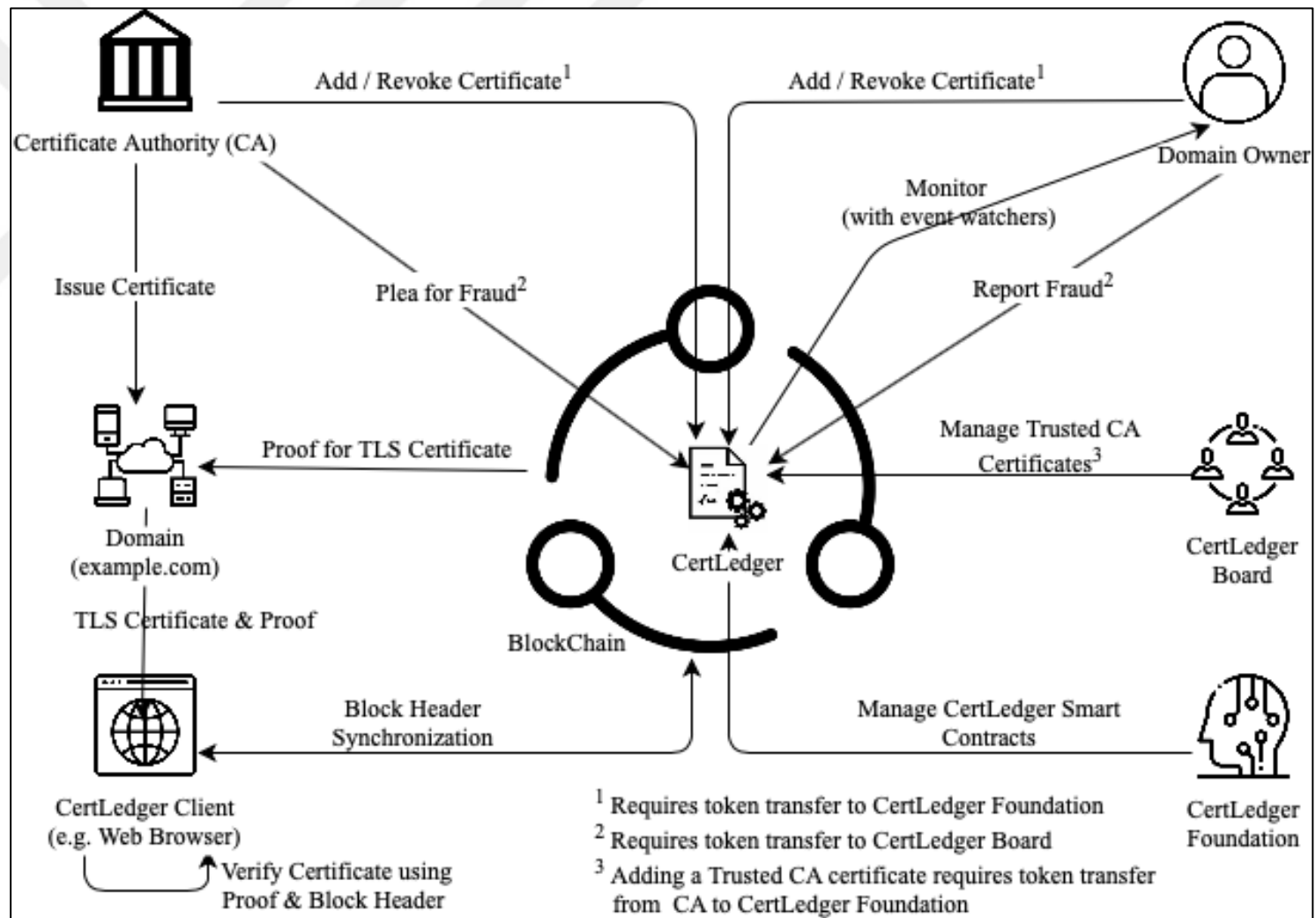


Figure 3.2: CertLedger Architecture.

- **CertLedger Token State Object:** is the source of the initial token supply. It comprises the smart contract code to determine the initial owner of the token supply, to transfer token between different Account State Objects and to give permission to a state object for transferring a certain amount of token from a given Account State Object.
- **Fraud Report State Object:** stores fraud reports about the CAs which are in the Trusted CAs State Object. The reports comprise proofs for the fraudulency of the CAs which possibly issued a valid but fake certificate. However, the accused CAs can also add proofs about their trustworthiness.

3.2.1. Entities

We have three types of entities in our model:

- i) **CertLedger Entities** (CertLedger Board, CertLedger Foundation, CertLedger Clients (TLS Clients))
- ii) **External Entities** (Certificate Authorities (CA), Domain Owners)
- iii) **Underlying Blockchain Entities** (Miners and Full Nodes).

3.2.1.1.CA

CAs have basically four different tasks:

- i) Check the identity of the Domain Owner for his TLS certificate request.
- ii) Issue a TLS certificate upon successful identity verification.
- iii) Optionally, create a transaction to append the issued certificate to CertLedger.
- iv) Optionally, create a transaction to change the revocation status of the certificate in CertLedger. Therefore, in our model, CAs do not issue Certificate Revocation List (CRL) and give OCSP services anymore.

3.2.1.2.CertLedger Board

CertLedger Board is a trusted organization who manages Trusted CAs State

Object for a better trust distribution among multiple entities. This board basically defines the standards and procedures to manage this state object. CertLedger Board members manage the Trusted CAs State Object by employing a t -out-of- l threshold mechanism (which is fault tolerant against arbitrary behavior of up to t malicious and colluding authorities) [77], [78]. More concretely, let $(PKBoard, (SKBoard_i, \dots, SKBoard_l))$ be the public and private key pair for the Account State Object of CertLedger Board where $PKBoard$ denotes the public key of the board members and $SKBoard_i$ denotes the private key share of the i -th member.

$$SKBoard := Construct(SKBoard_1, \dots, SKBoard_l) \quad (3.4)$$

is not known by anybody. $PKBoard$ is embedded in the smart contract code of the Trusted CAs State Object and is used to verify the transactions which are targeted to this object. When the CertLedger Board changes due to introducing new members, or quitting membership, $PKBoard$ and the $SKBoard$ also changes. Therefore, Trusted CAs State Object have to be migrated, so that it can comprise the new $PKBoard$.

In order to provide transparency, the requirements of being a board member should be defined as an international standard by the organizations such as IEEE [80] and IETF [81]. To decentralize the trust, the standards should enforce the selection of members from politically and geographically disparate entities (e.g., certificate authorities, browsers and O/S development companies/foundations, research institutes, universities, and international standardization organizations like ETSI [34] and IETF [81]) can be CertLedger Board members, but any organization in the world fulfilling the requirements can be in principle a board member.

3.2.1.3. CertLedger Foundation

CertLedger Foundation promotes, supports, and develops CertLedger platform and does research activities. They are also the owner of the initial CertLedger token supply.

3.2.1.4.CertLedger Clients

CertLedger Clients are TLS clients which are also part of the blockchain network and communicate with their peers for the following purposes. Using the light client protocol of the blockchain network, they fetch, validate, and store the block headers. They also download a Merkle state proof to validate the final state of a state object from a Full node. These proofs are later used to verify the TLS certificates of the domains.

3.2.1.5.Miners

Miners are writers which select pending transactions from the pool, validate them, and then create new blocks according to the consensus protocol. They generate blocks for all the transactions of the underlying blockchain and propagate them to the P2P network.

3.2.1.6.Full Nodes

Full Nodes store, validate, and propagate all the blocks and generate (Merkle) proofs for the CertLedger Clients to verify the final state of the state objects.

3.2.1.7.Domain Owners

Domain Owners offer secure services to the CertLedger Clients through protocols such as https, imaps, and sips. They have the following tasks:

- i) Make TLS certificate request to CAs for their domain.
- ii) Optionally, create a transaction to append the received TLS certificate to CertLedger.
- iii) Monitor their up-to-date TLS certificate in CertLedger (e.g., using event listeners on Ontology).

- iv) In case of compromise detection, immediately create a transaction to revoke their TLS certificate and create another transaction to report the fraud.

3.3. Trust and Threat Model

CAs are assumed to be malicious which can behave arbitrarily, e.g., fake but valid certificates can be issued by the CAs which may be either corrupted or operated with inadequate security policies. We also assume that the underlying blockchain network of CertLedger is insecure where a certain number of Miners are honest with respect to the underlying consensus algorithm for the agreement on the up-to-date state of the blockchain. Similarly, full nodes are also assumed to be malicious which are allowed to generate fake blocks and proofs. CertLedger Clients are also assumed to be malicious. Finally, at most t -out-of- l CertLedger Board Members are assumed to be corrupted. Trivially, we assume also that the underlying cryptographic primitives of the blockchain architecture of CertLedger are secure.

3.4. PKI Functionalities of CertLedger

In this section, we describe the functionalities of CertLedger. The responsibility of the entities in the following stages are illustrated in Figure 3.3.

3.4.1. Adding a New Trusted CA Certificate.

CertLedger Board adds a CA certificate to the *Trusted CAs State Object* as follows (Figure 3.4 and 3.5):

- i) To apply for an audit, a CA creates a transaction and triggers CertLedger Token State Object to give allowance to Trusted CAs State Object for transferring token from its Account State Object to the CertLedger Board's Account State Object.
- ii) Upon application of the CA, CertLedger Board audits the CA and verifies whether it complies with the Trusted CA standards.

- iii) Upon a successful audit, it generates a transaction comprising the CA certificate to add the CA certificate to the Trusted CAs State Object, which is signed in a threshold fashion [77], [78].
- iv) Smart contract code in the Trusted CAs State Object verifies the signature of the transaction, and makes all the necessary checks on the CA Certificate.
- v) Upon successful validation of the CA certificate it is added to the Trusted CAs State Object as a trusted certificate.



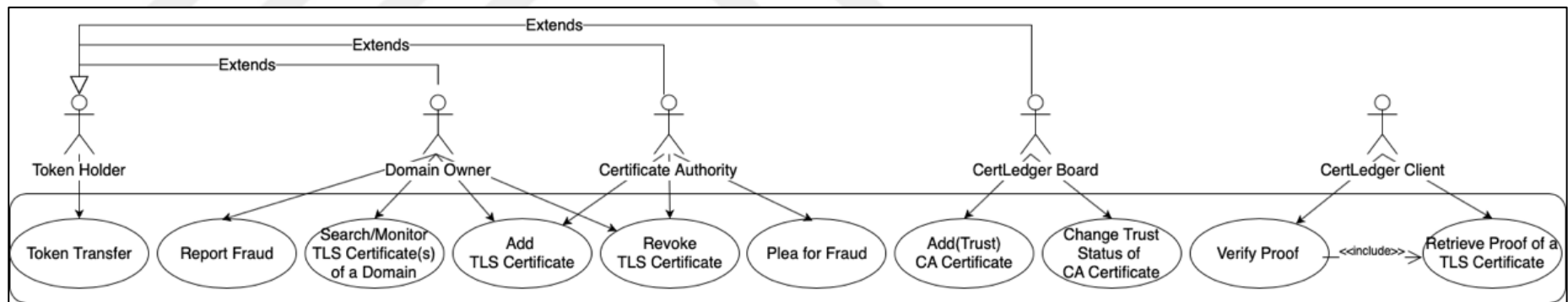


Figure 3.3: Functionalities of CertLedger.

- vi) Trusted CAs State Object triggers CertLedger Token State Object to transfer the operation fee from CA's Account State Object to the CertLedger Board's Account State Object.

```

Input:  $M := (Addr_{Board}, Addr_{TrustedCASO}, certCA), PK_{Board}, SK_{Board_{t_i}}$ 
/*  $Addr_{Board}$  is the AccountStateObject address of CertLedgerBoard,
    $Addr_{TrustedCASO}$  is the address of the TrustedCAsStateObject,  $certCA$  is
   the trusted CA certificate,  $PK_{Board}$  is the public key of the board,  $Board_{t_i} \in BoardList$ 
   involve in a threshold signing protocol to generate a transaction
   signature where  $1 \leq t_i \leq \ell, i = 1, \dots, t$ ,  $SK_{Board_{t_i}}$  is the private key share of
   the  $t_i$ -th board member */
Output:  $trx := (M, Sign_{SK_{Board}}(M))$ 
 $PSign_{t_i} := Sign_{SK_{Board_{t_i}}}(M) \forall i := 1, \dots, \ell$  //  $Board_{t_i}$  computes and publishes its
partial signature
 $Sign_{SK_{Board}}(M) := Construct(PSign_{t_1}, \dots, PSign_{t_t})$  // Any party can collect the
partial signatures and construct the final transaction signature.
if  $verify(Sign_{SK_{Board}}(M), PK_{Board}) = true$  then
|    $trx := (M, Sign_{SK_{Board}}(M))$  // Construct the transaction if the signature is
|   valid.
end

```

Figure 3.4: Generating a Transaction for Adding a Trusted CA Certificate through Board Members Algorithm.

```

Input:  $trx, certCAList := \{(SKI_j, certCA_j, ts_j) : 1 \leq j \leq m\}, PK_{Board}$ 
//  $SKI_j$  denotes for the  $j$ -th subjectKeyIdentifier field,  $certCA_j$  the  $j$ -th CA
certificate, and  $ts$  its trustStatus ("trusted" or "untrusted").
Output:  $\perp$  OR  $(certCA[SKI], certCA, "trusted") \in certCAList$ 
if  $trx.M.receiver = addr_{TrustedCASO}$  then
|   if  $verify(trx.signature, PK_{Board}) = true$  then
|   |   if  $certCA \notin certCAList$  s.t.  $certCA[SKI] = certCAList[SKI_j]$  for some  $j$  AND
|   |    $certCA[basicConstraints.CA] = true$  AND
|   |    $certCA[subject] = certCA[issuer]$  AND
|   |    $certCA[validity.notAfter] > t_{now}$  AND
|   |    $certCA[keyUsage] = "CertificateSigning"$  AND
|   |    $verify(certCA[signature], certCA[PK]) = true$  //  $certCA := trx.M.data,$ 
|   |    $certCA[PK]$  denotes the public key of  $certCA$ 
|   |   then
|   |   |   add  $(certCA[SKI], certCA, "trusted")$  to  $certCAList$ 
|   |   end
|   end
end

```

Figure 3.5: Adding a Trusted CA Certificate Transaction through Trusted CAs State Object Algorithm.

3.4.2. Changing Trust Status of a CA Certificate.

If a trusted CA fails to comply to Trusted CA standards in further audits or its misbehavior is proved through a fraud report, then the status of its certificate is set as “untrusted” in the Trusted CAs State Object as follows:

- i) CertLedger Board Members generate a transaction which is signed in a threshold fashion to change the status of the CA certificate in the Trusted CAs State Object.
- ii) Smart contract code in the Trusted CAs State Object verifies the signature of the transaction, and checks whether the CA certificate is unexpired and exists in the *Trusted CAs State Object*.
- iii) The state of the CA certificate is set as “untrusted” in the Trusted CAs State Object.
- iv) Trusted CAs State Object triggers Domain State Object to change the state of all TLS certificates issued from this CA as “revoked”.

Input: $trx, certCAList, certListURI := \{(uri_i, < certTLS_{i,j}, rs_{i,j} >) : 1 \leq i \leq \alpha, 1 \leq j \leq \beta\}, certListHash := \{(hash_k, certTLS_k, rs_k) : 1 \leq k \leq \gamma\}$ // uri_i is the i -th certificate subject alternative name, $certTLS_{i,j}$ and $rs_{i,j}$ is the j -th certificate and revocation status of the i -th uri resp., and rs_k is the revocation status of the k -th certificate $certTLS_k$.

Output: $\perp$ OR $((certTLS[subjectAlternativeName], certTLS, "valid") \in certListURI$
AND $(H(certTLS), certTLS, "valid") \in certListHash)$

```

if  $trx.M.receiver = addr_{DomainStateObject}$  then
  if  $verify(trx.signature, trx.M.PKSender) = true$  then
    if  $(certTLS[validity.notBefore] < certTLS[validity.notAfter])$  AND
        $certTLS[validity.notAfter] < t_{now}$  AND
        $certTLS[basicConstraints.CA] = false$  AND
        $certTLS[keyUsage] = "digitalSignature"$  AND
        $certTLS[extendedKeyUsage] = "serverAuthentication"$  AND
        $certTLS[subjectAlternativeName]$  is a valid URI AND
        $\exists certCA_j \in CAList$  s.t.  $certTLS[AKI] = certCA_j[SKI]$  for some  $j$  AND
        $certTLS[issuer] = certCA_j[subject]$  AND
        $certTLS[validityPeriod] \in certCA_j[validityPeriod]$  AND
        $verify(certTLS[signature], certCA_j[publicKey]) = true$  //  $certTLS := trx.M.data$ 
    then
      add  $(certTLS[subjectAlternativeName], certTLS, "valid")$  to  $certListURI$ 
      add  $(H(certTLS), certTLS, "valid")$  to  $certListHash$ 
    end
  end
end
end

```

Figure 3.6: Adding a TLS Certificate through DomainStateObject Algorithm.

3.4.3. Adding a New TLS Certificate.

As in the conventional PKI architecture, upon application of a Domain Owner, CA performs the relevant verifications according to its policy and issues the TLS certificate. Note that a Domain Owner can have more than one valid TLS certificate of a domain. In practice, before the expiration of his TLS certificate, the Domain Owner receives a new TLS certificate from the CA with overlapping dates, so that the latter becomes active before the expiration of the former. Upon generation of a new TLS certificate, Domain Owner or CA adds it to the Domain State Object as follows (Figure 3.6):

- i) Domain Owner/CA creates a new transaction comprising the new TLS certificate and signs it using the private key of his own Account State Object.
- ii) Smart contract code in the Domain State Object validates the certificate and whether the Domain Owner/CA's balance is sufficient for invoking the transaction. Upon a successful validation, the new TLS certificate is added to the Domain State Object.
- iii) Domain State Object triggers CertLedger Token State Object to transfer the operation fee from Domain Owner/CA's Account State Object to the CertLedger Foundation's Account State Object.
- iv) Domain State Object triggers an event notification after addition of the new certificate.

3.4.4. Revocation of a TLS Certificate

A TLS certificate is revoked in CertLedger as follows (Figure 3.7):

- i) A valid revocation request can only be generated for non-expired and valid TLS certificates. In our model, Domain Owners and CAs are assumed to be the only parties creating a valid revocation request. The revocation request of a certificate comprises a signature generated by either itself or by its issuing CA's certificate. The transaction comprising the revocation request is signed by the private key of the related entity's Account State Object. More concretely, the data in a transaction *trx* is formed as

$$\begin{aligned} \text{trx}.M.data := \\ (\text{revocationMessage}, \text{Sign}_{SK_{\text{certTLS}}}(\text{H}(\text{revocationMessage}))) \end{aligned} \quad (3.5)$$

where $\text{revocationMessage} := (\text{certTLS}, \text{'revoked'})$.

- ii) Smart contract code in the Domain State Object validates the transaction if the user's balance is sufficient for invoking the transaction. If the revocation code succeeds, then the state of the certificate is changed as "revoked" in the Domain State Object.
- iii) Domain State Object triggers CertLedger Token State Object to transfer the operation fee from Domain Owner/CA's Account State Object to the CertLedger Foundation's Account State Object.
- iv) *Domain State Object* triggers an event notification after revocation of the certificate.

3.4.5. Transferring Token.

Any entity having an Account State Object can transfer token ownership. While entities transfer token to be able to trigger some of the functionalities of CertLedger, they can also transfer token for only trading purposes. Token ownership is transferred as follows:

- i) An entity creates a transaction comprising the amount of the token to be transferred and the destination Account State Object address.
- ii) He signs the transaction with the private key of his Account State Object.
- iii) The transaction triggers the CertLedger Token State Object.
- iv) Smart contract verifies the signature of the transaction, and whether the balance of the entity is sufficient to make the transfer, and transfers the token from the sender's account to the recipient's account.

```

Input:  $trx$ ,  $certCAList$ ,  $certListURI$ ,  $certListHash$ 
//  $trx.M.data := (certTLSToBeRevoked, revocationMsg, signature)$  where
//  $revocationMsg := \text{"revoked"}$  and
//  $signature := Sign_{SKCertTLS}(H(certTLSToBeRevoked, revocationMsg))$ 
Output:  $\perp$  OR  $((certTLS[subjectAlternativeName], certTLS, \text{"revoked"}) \in certListURI$ 
AND  $(H(certTLS), certTLS, \text{"revoked"}) \in certListHash)$ 
//  $certTLS := trx.M.data.revocationMsg.certTLSToBeRevoked$ 
if  $trx.M.receiver = addr_{DomainStateObject}$  then
| if  $verify(trx.signature, trx.M.PKSender) = true$  then
| | if  $verify(trx.M.data.signature, certTLS[PK]) = true$  AND
| |  $trx.M.data.revocationMsg = \text{"revoked"}$ 
| | // the domain owner is requesting the revocation
| | then
| | | Change  $rs$  of  $certTLS$  in  $certListURI$  &  $certListHash$  as  $\text{"revoked"}$ 
| | else if  $\exists certCA_j \in CAList$  s.t.  $certTLS[AKI] = certCA_j[SKI]$ 
| | for some  $j$  AND  $verify(trx.M.data.signature, certCA_j[PK]) = true$  AND
| |  $trx.M.data.revocationMsg = \text{"revoked"}$ 
| | // the issuing CA is requesting the revocation
| | then
| | | Change  $rs$  of  $certTLS$  in  $certListURI$  &  $certListHash$  as  $\text{"revoked"}$ 
| | end
| end
end

```

Figure 3.7: Revoking a TLS Certificate through DomainStateObject Algorithm.

3.4.6. Secure Communication Through TLS

We assume that most of the CertLedger Clients are integrated into the internet browsers (or other applications using TLS) and run as a daemon. They receive only the block headers from the blockchain network when the browsers are up and running. During a TLS handshake, a CertLedger Client and the TLS server first agree upon the latest block number in the ClientHello and ServerHello messages through new TLS extensions (see Figure 3.8).

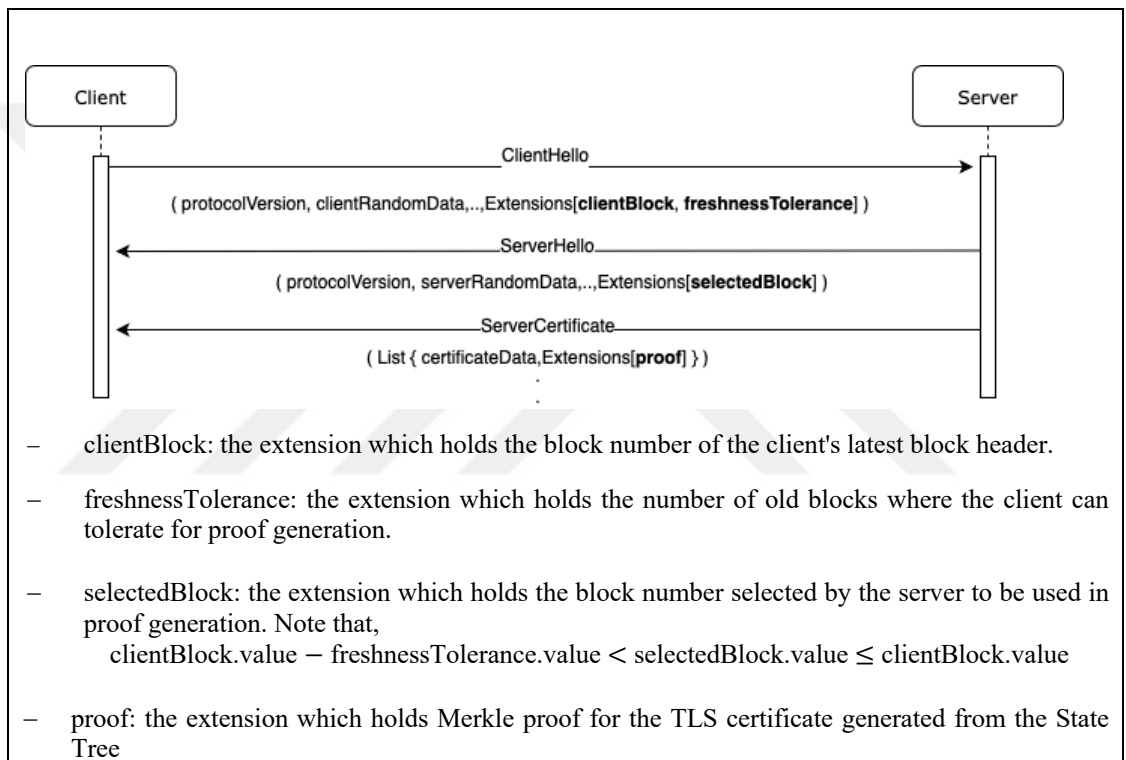


Figure 3.8: TLS Handshake Extensions for CertLedger.

The clients and the TLS server might have different up-to-date blocks due to network latency. Therefore, a client needs to tolerate up to certain number of old block and accept it as valid. More concretely, assume that the client has n th and the TLS server has $(n - i)$ th blocks. If the client tolerates upto j blocks then we must have $j > i$ for a successful handshake. CertLedger Client users can configure the level of j according to their security requirements. After a successful agreement, domain sends the *Domain State Object* of the TLS certificate, and its Merkle proofs generated out of the state

tree to the CertLedger Client in a TLS extension. If the domain does not have a Merkle state proof for its TLS certificate for the agreed block, then the TLS handshake aborts. The CertLedger Client checks whether the TLS certificate is issued for the domain, and is in its validity period. It verifies the proof using the state tree hash which exists in the agreed block header. It also checks whether the proof is indeed generated for the certificate and the state of the certificate is “valid”. We highlight here that CertLedger Clients do not require to make further validation of the TLS certificate, since this process is already completed while appending certificate to the blockchain.

We now elaborate the execution of TLS Handshake protocol in CertLedger. In the below steps $\mathcal{MMM}$: $\mathcal{N}$ denotes TLS message $\mathcal{MMM}$ comprising $\mathcal{N}$. $+ \mathcal{NNN}$ and $+ \mathcal{NNN}$ denotes a message sent as a standard or a CertLedger defined extension within the previous message respectively. $\{\mathcal{MMM}\}$ indicates a message $\mathcal{MMM}$ which is encrypted by AEAD with handshake traffic key tk_{hs} . $\mathcal{MMM}^*$ denotes a message sent only in unilateral mutual authentication modes.

Client

Server

ClientHello (CH):

$$r_c \leftarrow_R \{0,1\}^{256} \quad (3.6)$$

$$+ \text{ClientKeyShare (CKS): } X \leftarrow g^x \quad (3.7)$$

$$+ \text{ClientBlock (CB): } cb \leftarrow CB_{id} \in \mathbb{N}^+ \quad (3.8)$$

$$+ \text{FreshnessTolerance (FT): } ft \leftarrow i \in \mathbb{N}^+ \quad (3.9)$$

In CH, the client sends its supported TLS versions, cipher suites as well as the random nonce r_c to the server. The message contains the ephemeral Diffie-Hellman share g^x where g is the generator of the group G and x is the the private key of the client. Finally, CH also comprises the CertLedger extensions client block cb and the freshness tolerance ft .

ServerHello (SH):

$$r_s \leftarrow_R \{0,1\}^{256} \quad (3.10)$$

$$+ \text{ServerKeyShare (SKS): } Y \leftarrow g^y \quad (3.11)$$

$$+ \text{SelectedBlock (SB): } sb \leftarrow SB_{id} \in [cb, cb - ft) \quad (3.12)$$

In SH, the server sends client the random nonce r_s , its ephemeral Diffie-Hellman share g^y , and the selected block number sb as wells as its supported TLS versions and the cipher suites.

$$H_1 \leftarrow H(CH || SH) \text{ (including CKS\&CB\&FT\&SKS\&SB)} \quad (3.13)$$

$$ES \leftarrow Y^x \quad ES \leftarrow X^y \quad (3.14)$$

$$xES \leftarrow HKDF.Extract(0, ES) \quad (3.15)$$

$$tk_{hs} \leftarrow HKDF.Expand(xES, label_1 || H_1) \quad (3.16)$$

Both the client and the server calculates the ephemeral secret ES by calculating the Diffie-Hellman shared value g^{xy} . The key derivation is performed using extract-then-expand paradigm [97] which utilizes $HKDF$ to first compute the extracted value xES , then to expand traffic handshake key tk_{hs} from this value. Both xES and tk_{hs} are not authenticated at this stage of of the handshake.

{EncryptedExtensions (EE)}
{ServerConfiguration (SC)}

Starting from EE, the traffic is encrypted with tk_{hs} . Server sends more extensions in EE to the client, and a configuration which allows client an abbreviated handshake in SC.

{ServerCertificate (SCRT)}:

$$pk_s \quad (3.17)$$

$$\text{+Proof} : smp \leftarrow [(k_0 \in \mathbb{B}, v_0 \in \mathbb{B}), \dots, (k_i \in \mathbb{B}, v_i \in \mathbb{B})] \quad (3.18)$$

{CertificateRequest* (CR)}

SCRT contains pk_s and smp which denotes the public key certificate of the server and the merkle proof for the state of the pk_s for the block sb respectively. CR message is sent only in case the server authenticates the client through its certificate.

$$H_2 \leftarrow H(CH || \dots || CR) \quad (3.19)$$

{ServerCertificateVerify* (SCV)}:

$$SCV \leftarrow \text{Sign}(sk_s, H_2) \quad (3.20)$$

In SCV, server signs the hash of all the messages in the handshake from CH to CR.

$$SS \leftarrow Y^x \quad SS \leftarrow X^y \quad (3.21)$$

$$xSS \leftarrow \text{HKDF.Extract}(0, SS) \quad (3.22)$$

$$H_3 \leftarrow H(CH || \dots || SCV) \quad (3.23)$$

$$FS \leftarrow \text{HKDF.Expand}(xSS, \text{label}_2 || H_3) \quad (3.24)$$

{ServerFinished (SF)}:

$$SF \leftarrow \text{HMAC}(FS, \text{label}_3 || H_3) \quad (3.25)$$

In SF, server extracts xSS from the static secret SS which is then used to derive the finished secret FS . Afterwards it calculates a message authentication code over the hash of the handshake messages from CH to SCV using FS and sends to the client.

$$1 \leftarrow \text{check Verify}(pk_s, H_2, SCV) \quad (3.26)$$

$$1 \leftarrow \text{check Verify}(mth_{sb}, smp) \quad (3.27)$$

$$1 \leftarrow \text{check SF} = \text{HMAC}(FS, label_3 || H3) \quad (3.28)$$

Client verifies the authenticity of SCV by pk_s . It also verifies the existence and revocation status of the pk_s in the CertLedger using smp and the mth_{sb} which denotes the merkle tree hash of the State Merkle Tree for the block sb . Finally, it verifies the authenticity of the SF by using FS .

{ClientCertificate* (CCRT)}:

$$pk_c \quad (3.29)$$

$$H_4 \leftarrow H(CH || .. || CCRT) \quad (3.30)$$

{ClientCertificateVerify* (CCV)}:

$$CCV \leftarrow \text{Sign}(sk_c, H_4) \quad (3.31)$$

$$H_s \leftarrow H(CH || .. || CCV) \quad (3.32)$$

{ClientFinished (CF)}:

$$CF \leftarrow \text{HMAC}(FS, label_4 || H_s) \quad (3.33)$$

CCRT comprises pk_c which denotes the certificate of the client. In CCV, client signs the hash of the handshake messages from CH to CCRT using sk_c which denotes the corresponding private key of pk_c . In CF, it generates a message authentication code over the hash of the messages from CH to CCV using FS and sends to the server.

$$1 \leftarrow \text{check Verify}(pk_c, H_4, CCV) \quad (3.34)$$

$$CF \leftarrow \text{check HMAC}(FS, label_4 || H_s) \quad (3.35)$$

Afterwards server verifies CCV using pk_C and the authenticity of the CF using FS .

$$mES \leftarrow HKDF.Expand(xES, label_5 || H_3) \quad (3.36)$$

$$mSS \leftarrow HKDF.Expand(xSS, label_6 || H_3) \quad (3.37)$$

$$MS \leftarrow HKDF.Extract(mSS, mES) \quad (3.38)$$

$$tk_{app} \leftarrow HKDF.Expand(MS, label_7 || H_s) \quad (3.39)$$

Now, both the client and the server calculate the master secret MS by extracting the expanded ephemeral and static secrets. The traffic encryption key tk_{app} is then calculated by HKDF expansion steps which comprises the hash value of the handshake steps H_s and MS .

3.4.7. Fraud Management

In the following subsections we describe how a fraud is detected, reported and how to plea against fraud charge in CertLedger.

3.4.7.1. Fraud Detection

CertLedger does not prevent issuance of fake but valid TLS certificates and require monitoring. However, CertLedger provides infrastructure in order to detect fake TLS certificates and fraudulent revocations instantly. Mechanisms such as event watchers can be implemented which listen CertLedger events and inform a Domain Owner through several means (e.g., e-mail, SMS) upon a change on the state of his TLS certificates. The events in smart contract based blockchain architectures can easily be watched through APIs such as web3 [82] for Ethereum. Therefore, in CertLedger, monitoring process is effortless and the attack duration can be minimized due to promptness.

3.4.7.2.Reporting a Fraud

The fraud reports must also be transparent, and should not be submitted and processed in conventional mechanisms (e.g., email or personal applications). If the Domain Owner detects a fake but valid TLS certificate for his domain in CertLedger, he reports the fraud in CertLedger as follows:

- i) Domain Owner generates a transaction comprising the fake but valid TLS certificate and a signature generated with his genuine TLS certificate of the domain for proving his ownership to the domain.
- ii) He signs the transaction with the private key of his Account State Object and triggers the smart contract code of the Fraud Report State Object.
- iii) In the smart contract code, the existence of the fake but valid TLS certificate in CertLedger is verified, subject alternative names in the genuine and the fake TLS certificates are cross-checked, and the genuinity of the TLS certificate used for generating the signature in the transaction is verified.
- iv) If the contract code succeeds, the fraud report is added to the Fraud Report State Object.
- v) The Fraud Report State Object triggers the CertLedger Token State Object to transfer the operation fee from Domain Owner's Account State Object to the CertLedger Board's Account State Object.
- vi) Fraud Report State Object triggers an event notification after adding the new fraud report.

3.4.7.3.Pleading Against a Fraud

CertLedger Board and the Trusted CAs continually monitor the Fraud Report State Object through event watchers. A CA can plead in CertLedger, when it catches a notification triggered from the Fraud Report State Object which comprises a fraud charge against it as follows:

- i) Upon emergence of a new fraud report, it immediately puts all the necessary documents for the issuance of the TLS certificate to a portal of the CertLedger Board.
- ii) Generates a transaction comprising the hash of these documents signed with its CA certificate.
- iii) Signs the transaction with the private key of his Account State Object and triggers the smart contract code of the Fraud Report State Object.
- iv) Smart contract code verifies whether the plea is signed by the issuer of the fake TLS certificate and the CA certificate exists in the Trusted CAs State Object.
- v) The Fraud Report State Object triggers the CertLedger Token State Object to transfer the operation fee from CA's Account State Object to the CertLedger Board's Account State Object.
- vi) Fraud Report State Object triggers an event notification after adding the new fraud report.

Upon generation of the new block comprising this transaction, CertLedger Board makes a decision whether there exists a fraud. If the CertLedger Board is not convinced by the plea, then it creates a transaction to change the status of this CA certificate as “untrusted”.

3.5. Security Analysis

In this section, we analyse the security of CertLedger. We start by proving that it is resistant to split-world attacks and provides certificate transparency simultaneously. Second, we analyse the security of the TLS handshake by considering trusted CA certificate management, certificate revocation and validation processes. The comparison of CertLedger with the existing proposals from both security and privacy perspectives are given in Table 3.1.

Theorem 1. Assume that CertLedger is deployed on a blockchain which utilizes PBFT as the consensus algorithm (i.e., at most n out of $3n + 1$ consensus nodes are assumed to be corrupted). Then, CertLedger eliminates the split-world attacks if the underlying blockchain securely realizes the consensus.

Proof. Split-world attacks are only applicable if an attacker can present different views of the logs to the targeted victims. A blockchain node (either full or light node) has to query at least $2n + 1$ consensus nodes to be confident for the genuineness of a block. Split-world attacks are only applicable to CertLedger, if at least $n + 1$ consensus nodes are corrupted and they all cooperate with each other to generate fake blocks and construct a fake and hidden blockchain. These nodes can then convince targeted victims and propagate the fake blocks. For a successful split-world attack, they will also provide fake proofs to these victims. However, this assumption contradicts with the underlying consensus algorithm (i.e., at most n consensus nodes are corrupted).

□

If it is possible to identify the genuineness of a single block by a (t, l) threshold signature scheme between consensus nodes where at most t nodes are assumed to be corrupted, then a fake block can be generated either by at least $t + 1$ corrupted consensus nodes or by a vulnerability in the underlying threshold signature scheme [77], [78]. In either case, it contradicts with the threshold assumption or the security of the underlying signature scheme. Note that if less than t nodes are corrupted, fake blocks (or block headers) cannot be generated and propagated due to the underlying threshold signature scheme.

Theorem 2. Untrusted CA certificates can only be added to CertLedger if either at least $t + 1$ out of l board members or $n + 1$ consensus nodes out of $3n + 1$ (due to the underlying PBFT consensus algorithm) are corrupted. Similarly, the status of a trusted CA certificate can be unfairly changed as “untrusted” if either at least $t + 1$ out of l board members or $n + 1$ consensus nodes are corrupted.

Proof. A fraudulent CA certificate can be added to the CertLedger or the status of a Trusted CA certificate can be unfairly changed as ‘untrusted’, if more than t board members are corrupted³. This contradicts with the underlying (t, l) threshold signature scheme [77], [78]. Similarly, if $n + 1$ consensus nodes are corrupted and cooperate with each other then they could arbitrarily behave (e.g., generate fake blocks, fake board members). However, this assumption also contradicts with the underlying consensus algorithm where at most n consensus nodes are assumed to be corrupted.

□

Fraudulence and plea reports about the Trusted CAs are also managed in the CertLedger. Any Domain Owner who has detected a fake but valid TLS certificate, can put all the fraudulence proofs to the CertLedger. The accused Trusted CA can also put his plead transactions to the CertLedger as well. Upon inspection of the proofs, CertLedger Board Members can change the status of the Trusted CA as ‘untrusted’. In this respect, these kinds of CertLedger operations can also be publicly verified which makes it more transparent and reliable.

Property 1. The revocation process is completely conducted on CertLedger. CAs are only involved in the revocation process if Domain Owners cannot use the private keys of their certificates. Hence, it reduces dependency to CAs and revocation status of all TLS certificates can be tracked and verified transparently. To change the status of a TLS certificate as ‘revoked’, a revocation transaction comprising this certificate has to be included in a new block. The security of the revocation process in CertLedger is analysed under the following three scenarios:

³ In CertLedger, the requirements of being a Trusted CA will be defined in open standards. Only the certificates of the CAs complying these standards is assumed to be accepted as a Trusted CA certificate upon inspection of the CertLedger Board Members.

The private key is not compromised: If a Domain Owner cannot use the private key of his TLS certificate (e.g., forgotten password, malformed private key), then he will not be able to use this certificate to secure his service. Therefore, he has to use a new one for his domain. Since it has not been compromised, the revocation of an unusable TLS certificate would not also be necessary. To avoid unforeseen future incidents, the Domain Owner may simply destroy the hard/soft token of the private key.

The private key is compromised and the Domain Owner is aware of the compromise: If the Domain Owner can still use the private key of his TLS certificate, he immediately creates a revocation transaction without requiring the consent of the CA. On the other hand, if the Domain Owner cannot use his private key, then he would not be able to create a revocation transaction. In this case, he will immediately request it from the CA using conventional methods⁴.

The private key is compromised but the Domain Owner is unaware of the compromise: In this case, an adversary can have the following motivations: 1) To make impersonation/eavesdropping attack, 2) To interrupt the service of the domain. If the adversary has the first motivation without being detected, he should not create a revocation transaction. If his motivation is the latter, then he should create a revocation transaction. However, the revocation of the compromised certificate would also be appreciated by the Domain Owner, since he would be aware of the compromise, and immediately obtains a new certificate.

Property 2. Certificate validation is comprised of trusted path construction, validity checking, and privacy-preserving revocation status checking phases. CertLedger Clients securely verify the certificates during TLS handshake by eliminating the risks in these phases. Furthermore, privacy of CertLedger Clients is completely preserved.

⁴ As mentioned in [30], it is difficult to measure revocation reasons, since Domain Owners prefer to keep this information private. However, we foresee that this scenario would rarely happen due to the difficulty of compromising private keys in the hard tokens. Also, private keys in the soft tokens are backed up for recovery purposes.

Trusted Path Construction: In the existing systems, TLS clients have to store trusted CA certificates for trusted path construction. A TLS client has its Certificate/Key Store(s) which can be multiple with different contents depending on the applications and the internet browsers. However, management of Trusted CA certificates is a burden and one of the main sources of security compromise [83, [84]. CertLedger Clients do not need local Certificate/Key Stores anymore, since Trusted CA certificates are also managed in CertLedger. The trustworthiness of these certificates is analysed in Theorem 2.

Validity Checking: Improper implementations of TLS certificate validation by software vendors is one of the top ten OWASP security vulnerabilities in 2017 [85]. Namely, it is a common issue with one internet browser to be able to connect to a TLS protected domain while encountering connection problems with another browser due to the differences in certificate validation implementation. CertLedger eliminates this issue by validating TLS certificates prior to appending to the ledger according to the international standards. A CertLedger Client receives the certificate with its Merkle state proof from the domain during the TLS handshake. It verifies the proof using the state tree hash which exists in the block headers, and checks whether the proof belongs to the certificate and the certificate is in its validity period.

Privacy-Preserving Revocation Status Checking: CertLedger stores and manages the revocation status of the TLS certificates, therefore it eliminates the revocation services offered by CAs (CRL and OCSP), and avoids certificate revocation checking problems due to non-existent, improper or corrupted revocation services. A CertLedger Client receives the revocation status of the certificate during the TLS handshake and verifies it using the block headers. As the authenticity and the revocation status of a certificate is verified without requiring any further network connection during the TLS handshake, privacy of CertLedger Clients is fully preserved.

Table 3.1: Security Comparison of Log Based Approaches to Certificate Management.

	CT [24]	AKI [57]	ARPKI [58]	DTKI [54]	Wang et al. [62]	CertChain [64]	Yakubov et al. [63]	CertLedger [98]
Resilient to split-world/MITM attack	No	No	No	No	No	No	No	Yes
Provides revocation transparency	No	Yes	Yes	Yes	Partly ^a	Yes	Yes	Yes
Eliminates client certificate validation process	No	No	No	No	Partly ^b	No	Yes	Yes
Eliminates trusted key management	No	No	No	No	No	No	Yes ^c	Yes
Preserves client privacy	No	Yes	Yes	No	Yes	No	No	Yes
Require external auditing	Yes	Yes	Yes	Yes	No	No	No	No
Monitoring promptness	No	No	No	No ^d	No ^e	No	Yes	Yes

^aRelies on CAs and the revocation process is not transparent

^bRoot certificate validation is still necessary

^cEliminates trusted key management in TLS clients, but does not propose a solution for managing Trusted CA certificates in the blockchain

^dDo not make monitoring due to assuming all master certificates are genuine

^eBlockchain architecture is inadequate

3.6. Performance of CertLedger

3.6.1. Storage

According to Verisign domain name industry brief, there are about 3.3×10^8 registered domain names at the end of 2017 [86], and approximately half of them are using TLS certificates [87]. For calculations, we use the following facts. First, all TLS certificates will be based on elliptic curve cryptography (ECC). These certificates are expected to be dominantly used across the internet due to smaller key sizes, higher performance, and increased security. Second, the size of a TLS certificate which uses 256 bits of EC public key, signed with ECDSA signature algorithm, is approximately 2^9 bytes. Third, the maximum life time period of TLS certificates is specified as two years (825 days) in a recent CA/B Forum Ballot [88].

The size of the underlying blockchain of CertLedger depends on its block size and the frequency of adding new blocks. Block size increases with respect to the number of comprised transactions and the target block time depends on the selected consensus algorithm of the underlying blockchain (e.g., the target block time in Bitcoin PoW consensus algorithm is a trade-off between the propagation time of the new blocks and the amount of work wasted due to chain splits, and the target block time in the Ouroboros algorithm is a simple parameter that can be changed at any time according to the network efficiency [89], [52]). While making the calculations, for illustration purposes we select the target block time as 1 minute because

- i) The required storage capacity of the CertLedger Clients will increase with shorter block times where some of them have storage constraints and cannot be upgraded easily.
- ii) Selected block time should be short enough to discourage an adversary to make an attack.

Majority of transactions are expected to be about adding new certificates, therefore we ignore other transactions within CertLedger⁵. In this respect, storage space requirements of a Full Node and a CertLedger Client are detailed as follows:

Full Nodes : The number of transactions in a block is

$$BlockTotalTrx := TotalCert / AnnualGenBlock \quad (3.40)$$

where *TotalCert* and *AnnualGenBlock* denote the total number of TLS certificates and annually generated blocks, respectively. Similarly, the size of a block is

$$|Block| := |Header| + |trx| \times BlockTotalTrx \quad (3.41)$$

The size of a typical transaction for adding a TLS certificate is

$$|trx| := |Msg| + |Sgn| \quad (3.42)$$

where

$$|Msg| := |PKSender| + |Receiver| + |Data|. \quad (3.43)$$

If we assume that the creation of TLS certificates with 1 year average lifetime is dispersed homogeneously throughout the year and the block time is 1 min, then the number of blocks generated in a year would be 5.256.000 (*AnnualGenBlock* := $60 \times 24 \times 365$).

The blockchain size is then calculated as

$$|Blockchain| := AnnualGenBlock \times |Block|. \quad (3.44)$$

Using the EC certificates, the average size of *trx* is

⁵ Revocations are necessary in case of key loss or compromise, and the expected number of these transactions are very few. Moreover, unnecessary revocation transactions are not expected due to the cost of adding new TLS certificates. Therefore, revocation transactions are ignored in the calculations.

$$660 \text{ bytes } (\approx 64 \text{ byte} + 20 \text{ byte} + 2^9 \text{ byte} + 64 \text{ byte}), \quad (3.45)$$

and the average size of Block is

$$128 \text{ KB } (\approx 508^6 \text{ byte} + 660 \text{ byte} \times (1.65 \times 10^8 / 526000)) \quad (3.46)$$

Consequently, the annual average size of Blockchain is 512 GB ($5\,256\,000 \times 128\text{KB}$). Considering the fact that the price of a 1 GB disk storage is approximately 0.02 USD [90], the cost of storing CertLedger transactions for a full node is only about \$10,24.

CertLedger Clients: They do not store the blocks but only the headers, and their required disk space is directly proportional with the stored number of blocks. As Satoshi describes in [51], the required disk space to store block headers in Bitcoin for 10 minutes block interval time is 4.2 MB ($6 \times 24 \times 365 \times 508$) per year. A similar calculation for Ethereum with the 1 minute block internal time results in 128 MB ($60 \times 24 \times 365 \times 508$) of disk space per year. We expect the header size of the underlying blockchain of CertLedger will be similar to Ethereum. In this respect, for two years of period, CertLedger Clients will require approximately 256 MB of disk space for all TLS protected domains⁷. However, this is the maximum amount of data to be stored by the CertLedger Clients, we expect that it is sufficient to store the block headers for a much shorter period of time according to their security requirements.

3.6.2. TLS Communication

The size of the data stored in TLS extension (for Merkle Proof) is

$$\log n \times \text{state object size} + \log m \times \text{size}(\text{TLSCertificate}) \quad (3.47)$$

⁶ Block header size of Ethereum [32].

⁷ There are also other studies proposed for efficient light node clients in recent studies such as NIPoPoWs by IOHK [91] and FlyClient by Luu et al. [92]. For instance, FlyClient proposes, not only storing previous blocks hash in block headers, but also the root of a Merkle tree which commits to all blocks. Therefore, a logarithmic sized proof will be enough to verify whether a specific block is a part of the blockchain. Consequently, it is sufficient for the light nodes to store the head of the chain. They only require another Merkle proof to verify whether a transaction is included in the block.

where n is the number of state objects in the state tree and m is the number of certificates in the data storage tree⁸. We assume that there is a single Account State Object for each TLS protected domain, and therefore, the number of state objects is equal to the TLS protected domains [87]. Consequently, the total communication overhead of a TLS handshake is

$$(\log_2 1.65 \times 10^8) \times 16^9 \text{ byte} + (\log_2 1.65 \times 10^8) \times 2^9 \text{ byte} \approx 14 \text{ KB} \quad (3.48)$$

3.6.3. Scalability of the Blockchain

According to [93], in the Bitcoin network, the mean time for a node to see a block is 12.6 secs, and 95% of the nodes see the block within 40 secs. Moreover, it also states that for the blocks whose size is larger than 20 KB, the propagation duration increases 80 ms for each additional KB. While bitcoin block size is 1 MB, CertLedger has only 128 KB in case of 1 min block time. Therefore, the block propagation duration will be strictly less than 12.6 seconds if a similar network topology of Bitcoin is used in the underlying blockchain network of CertLedger. In fact, increase in the number of peers and the connectivity speed up the propagation. Since the number of TLS clients is much more than the number of peers in Bitcoin network, the propagation duration would be faster in CertLedger [98]. Hence, most of the peers of the blockchain network of CertLedger receive the new block header within block time. Consequently, CertLedger Clients can also reach the most up-to-date state of CertLedger blockchain within block time.

3.7. Implementation And Experimental Results

We developed a prototype of CertLedger on a private Ethereum network with a limited set of functionalities. In this prototype, our smart contract stores TLS

⁸ The key/value pairs stored in a state object is maintained in this Merkle tree. There is a separate storage tree for each state object. The root of this tree is inserted into the state objects maintained in the state tree.

⁹ Approximate size of an Account State Object in Ethereum.

certificates and their revocation status but does not validate them due to the constraints in certificate parsing¹⁰. We modified a sample TLS client and server to demonstrate TLS handshake and certificate validation with CertLedger. For this sample, we added CertLedger TLS extensions to Bouncy Castle JSSE provider [94]. To get and validate Merkle state proof for the TLS certificates we used Eth-proof node-js API [95]. We used a MacBook Pro with Intel Core i5 (2,3 GHz) CPU, 16 GB memory and macOS Mojave OS.

Table 3.2 TLS Handshake Performance Results

Number of TLS Certificates in the Blockchain	1	100	1.000	10.000	100.000	1.000.000
TLS Handshake Overhead (bytes)	1363	4043	5657	7060	8381	10787
Certificate validation duration (ms)	19,8	20,1	20,2	22,9	23,3	23,8

In this prototype, we mainly focused on evaluating TLS performance in terms of communication overhead and certificate validation duration. In experimental results (Table 3.2), we observe that the size of the Merkle proof with respect to the number of certificates in the blockchain is compatible with our theoretical analysis. Proof verification durations are very close to each other and do not change significantly with the increasing number of certificates. In the conventional PKI, the CRL size for median certificate is 51 KB [30] and OCSP query/response size is typically around 2 KB. Note that to make a proper revocation check, either an OCSP request has to be made or a CRL has to be downloaded for each certificate in the trust chain. Moreover, only retrieving an OCSP response often costs to a latency penalty under 250ms [106] without validating it.

3.8. Comparison

We now compare CertLedger with the other proposals according to following criterias which is tabulated in Table 3.3.

¹⁰ Note that validating a certificate is trivial and does not bring additional computational complexity.

3.8.1. Log Proofs

CertLedger provides proof for the existence and revocation status for all the TLS certificates as DTKI, CertChain, and Yakubov et. al.'s scheme [63]. CT provides proof only for the existence of a certificate in the log whereas AKI and ARPKI provide proofs only for the valid TLS certificates which are not revoked and expired. Wang et. al.'s scheme [62] provides existence proof for all of the TLS certificates, but may not provide a revocation proof if a CA does not issue correct CRL or gives OCSP service.

3.8.2. TLS Handshake Performance

In CT, the proofs are sent to the clients during TLS handshake. However, the received proof is not enough to make a successful TLS handshake since TLS clients have to check the revocation status of the TLS certificate by conventional methods like CRL or OCSP which can be cumbersome due to big CRL files and latency or interruption in OCSP services. Moreover, if the TLS clients make a prior audit to the logs to check the existence of the proofs, handshake duration increases. In DTKI, TLS clients have to make network connections to both MLM and the CLM to request the proofs. In Wang et. al.'s scheme [62], web servers send the certificate transactions and the Merkle proofs in TLS extensions to the browsers, and the browsers validate the transactions using the block headers. In addition, browsers with high security concerns may also spend time for revocation checking with conventional methods. In Yakubov et. al.'s scheme [63], TLS clients have to call either a smartcontract which performs all the necessary certificate validations from leaf to root or a web service which checks the validity of each certificate on the chain. In CertChain, clients have to construct the trust chain and validate the certificate signature. Afterwards, they have to call a bookkeeper to check the certificate's existence and its revocation status in the blockchain. CertLedger Clients receive approximately 14 KB of extra data to verify the final state of the TLS certificate from the domain during the TLS handshake. They do not to make any further network connections.

Table 3.3: Comparison of Log Based Approaches to Certificate Management.

	CT [24]	AKI [57]	ARPKI [58]	DTKI [54]	Wang et al. [62]	CertChain [64]	Yakubov et al. [63]	CertLedger [98]
External Dependency During TLS Handshake	Yes	Yes	Yes	Yes ^a	No	Yes	Yes	No
Factors Effecting TLS Handshake Performance	Certificate and proof validation, Revocation checking, Auditing (optional)	Certificate and proof validation, Occasional checks for the ILS root hash with the Validators	Certificate and proof validation	Certificate and proof validation, Connection to MLM and CLM	Blockchain update, Verification of the transaction and the proof	Certificate signature validation locally, Certificate validation through Bookkeepers	Certificate validation through Smart contract or the Web service	Data transmission overhead and verification of the Merkle audit proof
Existence of Logs with Different Contents	Yes	Yes	Yes	No	No	No	Yes	No
Necessity of External Auditing	Yes	Yes	Yes	Yes ^b	No	No	No	No

Table 3.3 Comparison of Log Based Approaches to Certificate Management (continued)

	CT [24]	AKI [57]	ARPKI [58]	DTKI [54]	Wang et al. [62]	CertChain [64]	Yakubov et al. [63]	CertLedger [98]
Necessity of External Monitoring	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes ^c
Does TLS Clients Require to Store Trusted Keys or Certificates of Logs ?	Yes	Yes	Yes	Yes	Yes	Yes	No	No
TLS Clients Require a Copy of the Log	No	No	No	No	Partly (block headers)	No ^d	No ^d	Partly (few block headers)

^aClients have to make standard certificate validation which may require external resources

^bMakes an unrealistic assumption that fake Master certificates cannot be issued, apparently not relevant for strong adversaries

^cMonitoring is efficient, it is possible to be aware of the TLS certificate state change promptly

^dTLS clients in Yakubov et al.'s Scheme and CertChain cannot verify certificate validation responses

3.8.3. Independent Logs

In CT, AKI and ARPKI, there can be independent logs which comprise different sets of TLS certificates. If a TLS client is trusting to a log and the visited domain's TLS certificate has not been appended to the log, then the TLS connection will be unsuccessful. For the CAs and the domain owners, it is impossible to know the TLS clients' set of trusted logs. Hence, ideally, CAs have to append their TLS certificates to all of the independent logs to eliminate the unsuccessful TLS connections. But the set of logs is not fixed and new logs can arise anytime. Appending TLS certificates to a changing set of logs is a burden for the CAs. Moreover, necessity of monitoring and auditing different logs is another overhead which can also lead to security compromise. However, in CertLedger, all of the TLS certificates are appended to one single log with multiple copies, and CertLedger Clients do not need a trusted key for the verification of the log proofs. Only this single log has to be monitored.

3.8.4. Log Availability

In CT, logs have to be audited and monitored continually for consistency and finding fake TLS certificates. In A(RP)KI, after each ILS update or a time out period, domain owners need to download proofs from the ILS. In DTKI, clients have to request proofs from the MLM, CLM or their mirrors during each TLS connection. Due to these reasons, all of these proposals require the logs to be permanently available which may lead to a single point of failure. However, CertLedger is maintained in a P2P network, and a CertLedger Client can receive the block headers from any node of the blockchain network. Therefore, it is not dependent to a single or a set of log server.

3.8.5. Auditing and Monitoring

CT requires to be audited by external auditors such as TLS clients and monitored by the CAs and the domain owners. In AKI, there are Validators which check the consistency of ILSs and whether the TLS certificates are updated according to certificate policies. In ARPKI, Validators are optional and their role is distributed to other CAs or ILSs. DTKI relies on TLS clients for auditing, but do not require

monitoring due to existence of master certificates. Expiration of master certificates and issuance of fake master certificates by the compromised CAs are open issues in DTKI both related to monitoring. Wang et. al.'s scheme [62] does not require external auditing, however it does not propose an efficient monitoring architecture. CertChain, Yakubov et. al.'s scheme [63] and CertLedger do not require external auditing for checking its cryptographic consistency and behaviour since new blocks are verified and appended to the log only upon the consensus of the underlying blockchain network. Moreover, it can be monitored easily and promptly.

3.8.6. Key/Certificate Store Management

In CT, TLS clients have to store Trusted CA certificates and the public keys of the logs to validate the TLS certificates and the SCTs. Although, AKI states that installing the trusted CAs and ILSs certificates and public keys to TLS clients will be enough to make a successful TLS connection, TLS clients also receive signed validator information from the domains during the TLS handshake. In order to validate this information, clients must also trust the Validators' public keys. In ARPKI, TLS clients have to store the trusted CAs and ILSs certificates to validate the proofs received from the domain. In DTKI, TLS clients require only the MLM public key to verify the queried proofs during a TLS connection. CertChain and Wang et. al.'s scheme [62] clients continue to make the conventional trusted key/certificate store management. Yakubov et. al.'s scheme [63] does not propose any mechanism for trusted certificate management, and any entity can deploy its root CA certificate to the blockchain. However, Trusted CA certificates are stored and managed in CertLedger. While adding TLS certificates to CertLedger, they are verified whether they are issued by one of the Trusted CAs. Since CertLedger Clients do not make further verification, but only check the revocation status of the TLS certificate during TLS handshake, they do not require to store and manage any trusted key or certificate.

3.8.7. Privacy

In CT, TLS clients audit the log to ensure its consistency by requesting Merkle Audit Proof s for the SCT they have received from the domains during the TLS

connection. Moreover, they can use OCSP to check the revocation status of a TLS certificate. These processes enable logs and the OCSP servers, to trace the browsing history of the TLS clients. In AKI and ARPKI, domains send the proofs to the TLS clients, thus their privacy is preserved. In DTKI, TLS clients both query MLM and the CLM to receive proofs for the targeted domain during TLS handshake. It is proposed that, to preserve privacy, the domains can act as a proxy to make these queries. However, this proposal will increase the network latency even more in DTKI. TLS clients in Wang et. al's scheme [62] continue to make conventional revocation checking, hence their browsing history can be tracked by the OCSP servers. Yakubov et. al.'s scheme [63] and CertChain clients have to query a web server or a bookkeeper to validate the TLS certificates which breaches their privacy. In CertLedger, as discussed in Section 4.4, privacy of the clients is fully preserved since the Merkle state proofs are provided by Domain Owners.

4. KORGAN: AN EFFICIENT PKI ARCHITECTURE BASED ON PBFT THROUGH DYNAMIC THRESHOLD SIGNATURES

In this Section, we first revisit one of the most recent blockchain based proposals for PKI (i.e., the Yakubov et al.'s scheme), and address its security and privacy issues within their certificate validation phase during TLS handshake. More concretely, TLS clients can easily be deceived by fraudulent full nodes or web services during certificate validation, because they cannot verify the validity of the incoming responses. Besides, fake but valid certificates also cannot be detected since only the hash values of the certificates are stored in the blockchain. In order to eliminate these issues, we improve their scheme and propose a new PKI architecture what we called KORGAN. In summary, our scheme provides the following features.

- i) We use Practical Byzantine Fault Tolerance Algorithm (PBFT) [33] as the consensus mechanism where the consensus nodes hold a share of the blockchain signing key and a block can only be generated if a threshold number of them approve the block by signing it by their partial key share [35]. Since we use a dynamic threshold signature scheme, once the TLS clients receive the blockchain verification key (i.e., public key) they will not require to change it even if the set of validators is dynamic.
- ii) During TLS handshake, TLS clients can validate the certificates without requiring to be a peer of the blockchain network. Moreover, they do not need to make any further network connection and query other entities during this process. In this respect, certificate and revocation transparency is now fully provided so that the TLS certificates and their revocation status are publicly monitored. Moreover, the privacy of the TLS clients is fully preserved during certificate validation.
- iii) CAs are not the sole authority to revoke certificates anymore, certificate owners can now also revoke their certificates (if they still possess the private key). Therefore, the risk of not revoking a compromised certificate in a reasonable timeframe by a malfunctioning CA is minimised.

We implement a prototype of our proposal on Ethereum, and experiment certificate validation. Our experimental results show that TLS clients can validate certificates efficiently (in constant time) depending on only their processing power and memory. Moreover, TLS handshake overhead is insignificant in our scheme.

4.1. Security and Privacy Analysis of The Yakubov Et Al.'s Scheme

In this section, we first briefly describe the Yakubov et al.'s scheme and then point out its security and privacy issues. In this scheme, each CA has a dedicated smart contract for issuing and revoking certificates in the blockchain. More concretely, a CA smart contract contains the following features:

- $certificateList := \{ \langle index_i, certHash_i, hashAlg_i, date_i \rangle : 1 \leq i \leq \alpha \}$ where $index_i$ is the auto generated index for the i -th certificate, $certHash_i$ is the hash value of the i -th issued certificate using $hashAlg_i$, and $date_i$ is the addition date of the certificate to the blockchain.
- $revocationMap := \{ \langle index_i, revokeDate_i, hashAlg_i, date_i \rangle : 1 \leq i \leq \alpha \}$ where $index_i$ is the index of i -th certificate in the $certificateList$ and $revokeDate_i$ is the revocation date of the i -th certificate.
- Its CA certificate $Cert_{CA}$.

Once a certificate is issued, the CA adds the hash value of the certificate to $certificateList$. Similarly, to revoke a certificate, CA adds the index of the $certificateList$ and the revocation date to the $revocationMap$. The certificates in this proposal basically comprise several custom X.509 extensions such as CA key identifier and Issuer CA identifier:

- *CA key identifier* is populated with the CA smart contract address in the CA certificates.
- *Issuer CA identifier* stores the smart contract address of the issuer of a certificate. This extension is populated for all the certificates apart from the root CA certificates. In fact, it is used for building a trusted path and finding

the smart contract address of the issuer of a certificate during the certificate validation.

Certificate validation can be performed in two different methods.

- i) In the first scheme, the certificate validation algorithm is implemented in a smart contract. This smart contract triggers all the CA smart contracts in the trusted path of a certificate. It validates the existence and the revocation status of all the certificates within this path. This scheme can only be triggered through a full node of the blockchain.
- ii) In the second scheme, certificate validation is delegated to a web service. The web service queries the revocation status of all certificates in the trust chain one by one from a full node.

According to the experimental results, the performance of the second certificate validation scheme has a higher performance than the first one for the trust chains up to 400 sub-CAs. We depict these certificate validation schemes in a TLS system in Figure 4.1.

Fake but valid certificates cannot be identified: The transparency of the certificates is not fully provided in Yakubov et al.'s scheme since only *certHash_i* are stored in the CA smart contracts. Since it is infeasible to derive the subject of a certificate from *certificateList*, it would not be possible for a domain owner to identify and revoke a fraudulent certificate. Consequently, during TLS handshake, clients accept fake but valid certificates allowing MITM kinds of attacks.

Certificates may not be revoked on time in case of corrupted CAs: CAs may have to revoke their issued certificates for several reasons such as key compromise or information change (e.g., DNS name). However, if a CA is compromised or does not have a proper revocation process, the certificate could not be revoked in a reasonable time frame or could not be revoked at all. This CA dependent process makes the TLS clients vulnerable to MITM kinds of attacks between the compromise and the revocation time of the certificate.

Certificate validation services can be compromised: The proposed certificate validation schemes are subject to MITM kinds of attacks. This is because the TLS clients have to rely on either the full nodes or the web service, and trust their certificate validation responses. These responses do not contain any cryptographic proofs of correctness. Therefore, clients can easily be deceived if the full nodes¹¹ or web services are corrupted.

Building the trust chain of a certificate is one of the critical components of certificate validation. This process is not fully clarified in the proposal which may be a cause of MITM attacks as well. Namely, creation of CA smart contracts on the blockchain is not subject to authorization, therefore an adversary can deploy a smart contract for a malicious CA which stores fraudulent certificates issued by this fake CA. In this case, TLS clients will be subject to MITM kinds of attacks since they will validate these fake certificates through the proposed certificate validation schemes.

Privacy issue while certificate validation: TLS clients cannot validate the certificates themselves (trivially, if they are not a full node of the blockchain). In that case, they have to query a full node or a web service for this purpose. However, this process is not privacy preserving since these intermediary entities can track the web addresses visited by the TLS clients.

4.2. Our Proposal: KORGAN

We first describe the relevant blockchain architecture for our PKI model and propose our modified framework based on Yakubov et al.'s scheme to eliminate the aforementioned issues.

¹¹ Full nodes do not execute a transaction which changes the state of the blockchain while running certificate validation smart contract, but only queries blockchain data. Therefore, its malicious behaviour does not have any impact on blockchain.

4.2.1. Our Motivation: Why Dynamic Threshold Signatures Based Permissioned Blockchains?

In order to eliminate the security and privacy issues mentioned in the previous Section, TLS clients should be able to use a publicly available blockchain which would include all issued certificates as well as their status without relying any external parties during a certificate validation process. However, this introduces an extra overhead for both permissionless and permissioned blockchains since they have to first verify the validity of the blocks (or only the headers) [100]. More concretely,

- i) In case of permissionless blockchains, there is going to be a significant network overhead for TLS clients because they need to be a peer of the blockchain network to determine the valid blocks due to the underlying consensus mechanism [32], [51], [52].
- ii) In case of permissioned blockchains, they have to query a certain number of consensus nodes (i.e., $2N + 1$ in PBFT [33] which requires $3N + 1$ replicas to tolerate N Byzantine failures). This requirement would also become a burden with the increasing number of consensus nodes.

Therefore, becoming a peer of the blockchain network or querying consensus nodes would going to be infeasible for many of the TLS clients due to limited storage capacity, processing power, or low bandwidth. In order to eliminate this overhead, we require authentic blocks which would enable TLS clients to verify their validity efficiently. However, permissionless blockchains are unfortunately not suitable for generating signed blocks because any peer could join the blockchain network and could generate a new block which makes determination of a signature (private) key and distribution of the verification (public) key to the TLS clients infeasible. On the other hand, permissioned blockchains would be more suitable for generating signed blocks since only a limited number of consensus nodes are authorized to generate the new blocks. Still the following requirements must be satisfied for an efficient and scalable solution:

- i) Management of the verification key should not be a burden for the TLS clients. Namely, after they receive an authentic verification key, they should not be able to change it frequently.
- ii) There must be only one signature on a block to be optimally scalable.
- iii) The verification key should not be also changed with the varying number of consensus nodes (i.e., in case of adding new nodes or removing the existing ones). Otherwise, all TLS clients must subsequently update the verification key which would also make the system practically infeasible



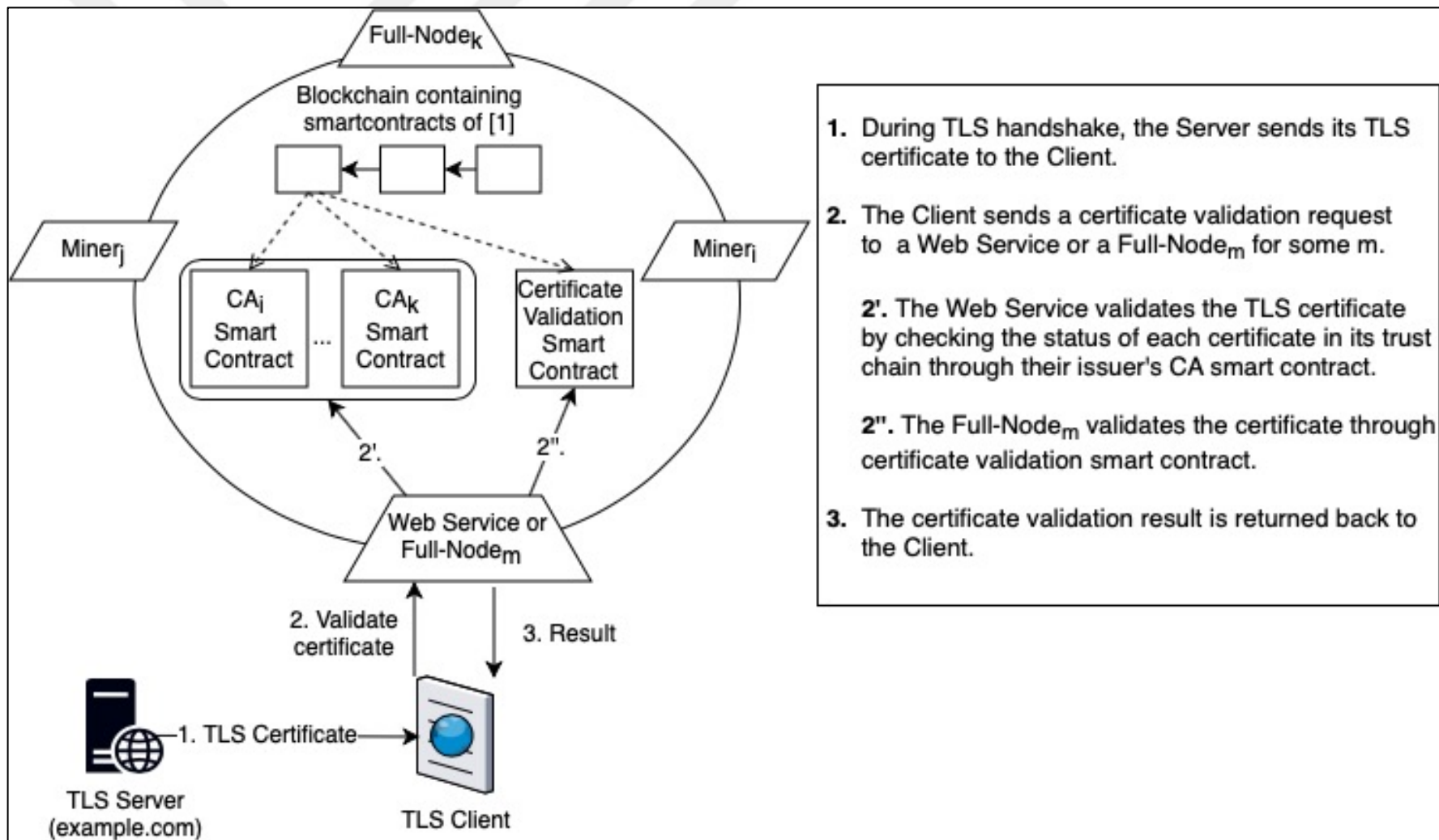


Figure 4.1: TLS System of Yakubov et al.'s Scheme.

To tackle these requirements, we propose to use dynamic threshold signature schemes among the consensus nodes for signing the new blocks [35]. As in a typical threshold signature scheme, there is going to be only one public key (for verifying a signature) of the overall system, and the private key shares will be owned and managed by the corresponding consensus nodes. If at least a threshold number of consensus nodes agree on a block, then they are generate a valid signature.

We highlight that the underlying consensus of Facebook Libra also utilizes threshold signatures [101]-[103], however, their solution does not propose dynamic versions of threshold schemes which would incurs significant overhead to our scheme. This is because it does not the meet the above-mentioned third requirement, and a change in the underlying consensus nodes would result in an update in the clients' public (verification) keys. Thanks to the authors of [35] we have efficient threshold schemes which indeed meet all the requirements and ensure that the remaining consensus nodes are able to add new consensus nodes or remove the corrupted nodes efficiently through only updating their secret shares without changing the overall verification key.

4.2.2. Our Approach: PBFT Consensus Mechanism with Dynamic Threshold Signatures

The seminal Practical Byzantine Fault Tolerance (PBFT) algorithm aims to reach consensus through Byzantine nodes that tolerates Byzantine failures with low overhead [33]. In particular, PBFT basically uses state-machine replication and replica voting for changing the state in the network. All nodes acting as validators¹² have equal votes, and validation is executed through multiple rounds to reach the consensus. PBFT utilizes digital signatures to ensure the authenticity of the messages. Nodes have to verify all the signatures received from their peers during each phase of the consensus rounds.

In our blockchain architecture, we utilize a dynamic threshold signature scheme on the PBFT consensus mechanism, and a valid block can only be generated if at least

¹² Trivially, as in any permissioned blockchain, we require the validators to be selected from political and geographical disparate entities.

t -out-of- l consensus nodes sign the new block [35]. Key generation setup for threshold signature scheme is going to be executed among predefined consensus nodes as follows:

4.2.2.1. Threshold Key Generation Setup Among Consensus Nodes

We assume that an existing permissioned blockchain with $3N + 1$ consensus nodes have been already setup [33]. Namely, the key generation ceremony will be completed using the underlying PBFT consensus mechanism. More concretely,

- i) Each i -th consensus node randomly chooses its secret key share SK_i and executes the threshold signature steps (like in [35]), and publishes their intermediate outputs on the blockchain (as a transaction). In particular, the consensus nodes use the underlying blockchain as a public bulletin board to publish and retrieve the necessary data to execute the key generation setup properly.
- ii) Each consensus node queries the blockchain until all the consensus nodes publishes their partial signatures. A key generation setup would be completed only after all the pre-defined consensus nodes participate to the ceremony¹³.
- iii) Once all the consensus nodes publish their partial signatures to the blockchain, the selected leader constructs the overall public key and adds as a new transaction. More concretely, once the signature steps are validated through the consensus mechanism, $(PK, (SK_1, \dots, SK_l))$ become the verification key and signing key share of the i -th consensus node, respectively.

Once the key generation ceremony is completed successfully, the consensus nodes (i.e., the signers) will only generate and accept signed blocks.

¹³ If they do not take part in the ceremony, they will not be able to send signed messages during the consensus phase.

4.2.2.2. Block Generation through PBFT with Dynamic Threshold Signatures

We now have a dynamic threshold signature scheme on top of PBFT. Note that the underlying consensus mechanism would run in rounds with one node acting as a leader and others as validators. In order to sign a new block, the consensus nodes (i.e., the signers) are going to execute the following threshold signature generation ceremony to generate a valid block.

- Pre-prepare phase: The goal of this phase is to ensure that a majority of honest nodes has agreed on a sequence number for a leader's request. In our flow, the leader initiates the consensus process by sending its partially signed PRE-PREPARE message (using $Sign_{SK_i}$) which is a block proposal containing a certain number of transactions.
- Prepare phase: Upon receiving the PRE-PREPARE message, every node in the consensus group checks the correctness and validity of the block and multicasts its partially signed PREPARE message (using $Sign_{SK_i}$) to all the other nodes.
- Commit phase: Based on the received PREPARE messages, each node combines t signatures, computes a valid signed message, and then multicasts a signed COMMIT message (i.e., "YES/NO") to the consensus group. The new block is committed to the blockchain only if a valid signature generated.

At the end of the commit phase, all honest nodes in the consensus group would have the same view regarding to the state of blockchain by either accepting or rejecting the block proposal, thereby achieving the confirmed transaction. Consequently, the authenticity of a block (or a block header) can easily be verified by any TLS client using the verification key PK . Since the underlying consensus mechanism relies on PBFT, up to N nodes could suffer from Byzantine failure. Therefore, $t = 2N + 1$ out of $3N + 1$ nodes would be sufficient to preserve the same security level of the underlying consensus mechanism.

4.2.2.3. Adding and Removing Consensus Nodes

Upon addition and removal of the consensus nodes, private key shares of all the consensus nodes has to be updated without changing the overall (PK, SK) as described in [35]. Furthermore, the threshold t has to be updated in such a way that it has to preserve the same security level of the underlying PBFT.

Assume that we have $3N + 1$ consensus nodes and $t = 2N + 1$. If we want to add (or remove) $3m$ nodes, then we have in total $3(N + m) + 1$ (or $3(N - m) + 1$ in case of removal) nodes. Then, t must become $2(N + m) + 1$ (or $2(N - m) + 1$ in case of removal) to reach the same security level of PBFT.

4.2.3. The CA Smart Contract in KORGAN

As in the Yakubov et al's scheme, in KORGAN, we assume that CAs validate the domains (i.e., ACME protocol [49] for DV SSL Certificates¹⁴) using the processes described in their certificate policy and certificate practice statement documents, and issue the certificate afterwards. In the following, we will describe the blockchain structure and the CA smart contracts in KORGAN.

We use the permissioned blockchain with the PBFT consensus mechanism (described in Section 4.2) which allows TLS clients to easily verify the final state of a certificate. To enable this, consensus nodes maintain a State Merkle Tree (SMT)¹⁵ which is used to store and verify the state of all accounts, smart contract codes and the data within the smart contracts. SMT is updated with each block according to the block transactions and its Merkle root is put into the generated block header¹⁶. Since we store all the issued certificates and their revocation status in the smart contracts, their state can also be tracked in every block. Moreover, their state can be verified using the Merkle proof generated from the SMT and the Merkle root. Since Merkle root is stored in the signed block header and its authenticity can be verified by PK , TLS clients can

¹⁴ Domain Validation SSL certificates are issued after proving the right to use the domain.

¹⁵ Modified Merkle Patricia Tree [104] can be used for this purpose where search, insert and update operations can be performed in logarithmic time.

¹⁶ A sample block header of our blockchain architecture is depicted in Figure 4.2.

easily check the validity of a certificate without relying any parties during TLS handshake.

Instead of an array with certificate hash values and a map for the revoked certificates, KORGAN modifies the CA smart contract in the Yakubov et al.'s scheme by storing the following two maps:

The first map is

$$certURIMap := \{(uri_i, < certHash_{i,j}, >: 1 \leq i \leq \alpha, 1 \leq j \leq \beta\} \quad (4.1)$$

where uri_i is the i -th certificate subject (or subject alternative name), $certHash_{i,j}$ is the hash of the j -th certificate of the i -th uri , and $\alpha, \beta \in \mathbb{N}^+$. This map provides the transparency of the certificates, and the domain owners can use it to monitor¹⁷ the blockchain if a fraudulent certificate is issued for their web servers.

The second map is used to track the revocation status of the certificates and represented as

$$certRevocationMap := \{< certHash_k, rs_k >: 1 \leq k \leq \gamma\} \quad (4.2)$$

where $certHash_k$ is the k -th certificate hash, rs_k is the revocation status (“revoked”, “valid”) of the k -th certificate, and $\gamma \in \mathbb{N}^+$.

In our CA smart contract, a certificate can only be added to the blockchain by its issuing CA. However, the status of the certificate can be changed as “revoked” by both its issuing CA and owner. A certificate owner can only trigger the revocation method of the smart contract if he can prove his ownership to the certificate. Therefore, this method requires a signature generated by the private key of the certificate.

Note that block confirmation time must be short enough to discourage adversaries to perform a MITM kind of attack during block time. In this respect, due to the underlying PBFT mechanism and dynamic threshold signature scheme, our consensus scheme provides high throughput and low transaction latency similar to LibraBFT [105] which meets our requirements.

¹⁷ Event listeners can be used for this purpose which triggers certain events (e.g., SMS, e-mail etc.) in case a certain condition is satisfied in the smart contract.

4.2.4. Certificate Validation of KORGAN in SSL/TLS

TLS clients are not required to be full or light nodes of the blockchain, thus do not have to retrieve any blocks (or headers) from the blockchain network. For certificate validation, they only need to store *PK* to verify the authenticity of the block headers and the trusted root CAs' smart contract addresses to construct the trust chain. On the other hand, TLS servers have to periodically retrieve the block headers and the Merkle proofs associated with their TLS certificate chain from a full node of the blockchain. The retrieval process is independent of the TLS handshake and can be conducted asynchronously.

In our proposal, the TLS handshake protocol has not been changed but we introduce new TLS extensions to be used during the *ServerCertificate* step of the protocol (Figure 4.2). In these extensions, a TLS server sends the latest block header, certificate chain, and a list of Merkle proofs for the revocation status of each certificate in the chain to the TLS client. A TLS client performs the following steps to validate the TLS certificate.

- i) Verifies the signature of the block header by *PK*, and reads the authentic block generation time and the SMT root from the block header.
- ii) Checks whether the block generation time is fresh enough according to its security settings. However, the acceptable freshness period should not be shorter than the block time and it should not reject the latest block header.
- iii) Checks whether the TLS certificate is issued from a trusted root CA by searching the smart contract address (*CA key identifier*) of the root CA in its trusted list.
- iv) Validates each certificate in the trust chain by validating the Merkle proofs using the certificate's hash value, its issuing CA's smart contract address (*Issuer CA identifier*) and the SMT root.
- v) Checks the revocation status of each certificate in the trust chain (except root CA certificate) and verifies that none of them are revoked.

The first and second step of our certificate validation algorithm is described in more detail in Figure 4.3, third step in Figure 4.4, and finally fourth and fifth steps in Figure 4.4 and 4.5. We would like to highlight that our algorithm does not require any

further network connections, thus the privacy of the TLS clients is also fully preserved. The execution time of the algorithm is not effected from the network latency and only depends on the processing capability of the TLS clients.



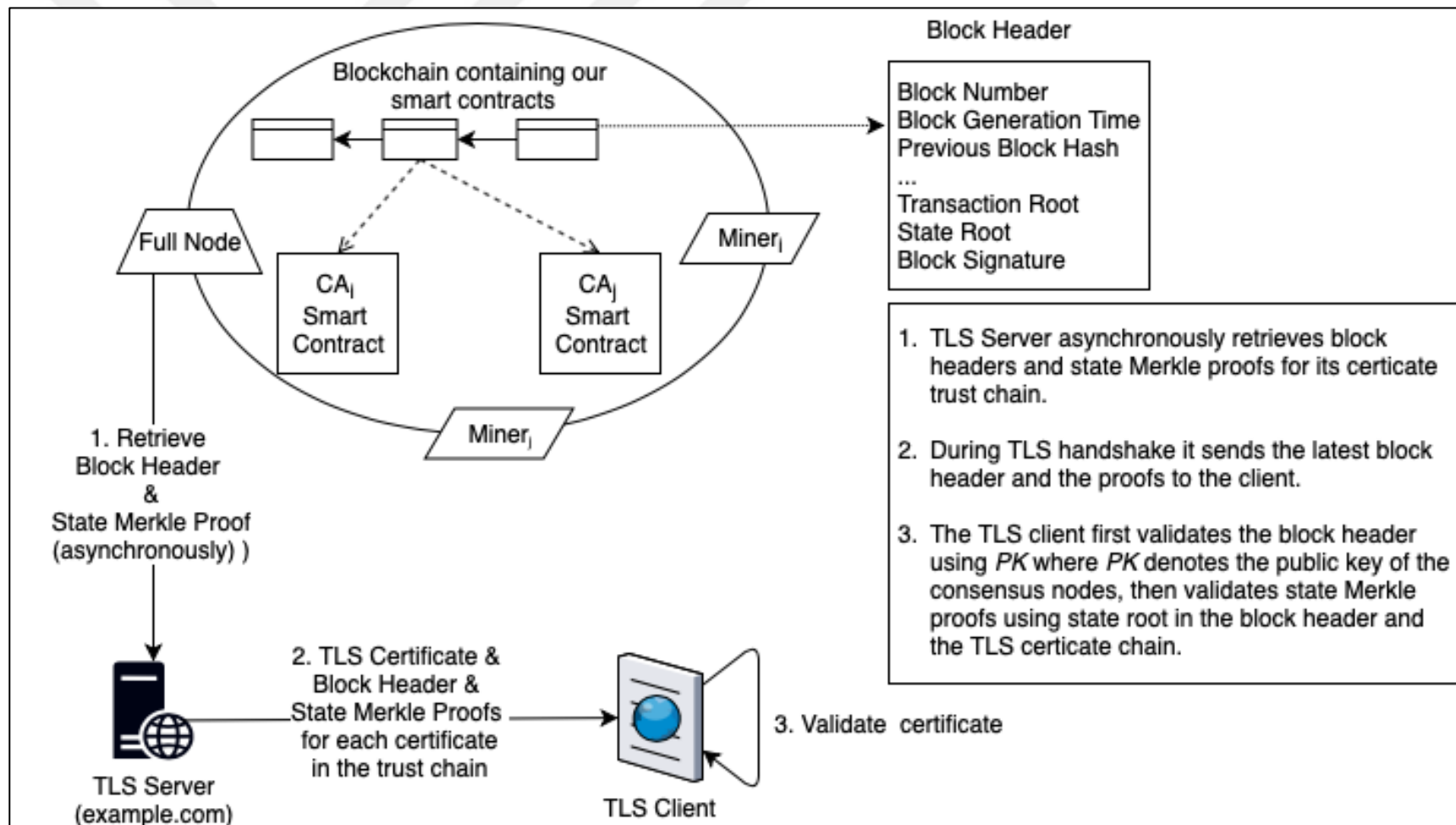


Figure 4.2: TLS System of KORGAN.

‣ *header* denotes the block header of the block which will be used to validate the state of the certificates, *PK* denotes the public key of the blockchain, *latestAcceptableTime* denotes the latest generation date of the *header* acceptable by the TLS client

```
function VERIFYHEADER(header, PK, latestAcceptableTime)
  ‣verify that the block header is genuine
  if verifyBCHeaderSgn(header.signature, PK) = false then
    return false
  end if
  ‣verify that the block header is not too old
  if (header.timestamp ≥ latestAcceptableTime AND
    header.timestamp < tnow) = false then
    return false
  end if
  ‣otherwise return true
  return true
end function
```

Figure 4.3: Verify Header Algorithm.



► *certChain* denotes the CA certificates in TLS certificate trust chain, *proofForCertStatusList* denotes the list of Merkle proofs for the status of each certificate in the trust chain except root CA certificate

function VALIDATECERTIFICATECHAIN(*certChain*, *proofList*, *trustedCAList*, *header*)

► verify that the root CA in the certificate chain is in the trusted CA address list

if *certChain*[*certChain.length*].*caKeyIdentifier* $\notin$ *addrTrustedCAList* **then**

return *false*

end if

► verify that the certificates in the TLS chain (except root CA certificate) are not revoked.

► note that if the certificates doesn't exist in the smart contract merkle proofs can not be verified

for *i* $\leftarrow$ 1 to *certChain.length* - 1 **do**

cert $\leftarrow$ *certChain*[*i*]

certCA $\leftarrow$ *certChain*[*i* + 1]

if ValidateCertificate(*cert*, *certCA*, *proofList*[*i*], *header.stateRoot*) = *false* **then**

return *false*

end if

end for

► otherwise **return** *true*

return *true*

end function

Figure 4.4: Validate Certificate Chain Algorithm.



► *cert* denotes the certificate to be validated, *certCA* denotes the certificate of *cert*'s issuer, *merkleProof* is the merkle proof generated from SMT for *cert*, *smtRoot* denotes the root hash value of SMT

function VALIDATECERTIFICATE(*cert*, *certCA*, *merkleProof*, *smtRoot*)

caScAddr $\leftarrow$ *cert.issuerCAIdentifier*

certHash $\leftarrow$ Hash(*cert*)

if *caSCAddr* $\neq$ *certCA.caKeyIdentifier* **then**

 ► check whether *certCA* is the issuer of *cert*

return *false*

end if

 ► verify that certificate is valid using the state proof generated for the certificate

if verifyStateMerkleProof(*merkleProof*, *smtRoot*, *certHash*, *caScAddr*) $\neq$ "valid" **then**

return *false*

end if

 ► otherwise return true

return *true*

end function

4.3. Security Analysis

Theorem. Assume that KORGAN is deployed on a blockchain which utilizes PBFT as the consensus algorithm (i.e., at most n out of $3n + 1$ consensus nodes are assumed to be corrupted) where the validators utilize dynamic threshold signature scheme and a valid block can only be generated if at least t -out-of- l validators sign the new block (i.e., $t=2N+1$ out of $l=3N+1$ nodes should be trustworthy). Then, the authenticity of a block can be verified by using the public key of the consensus nodes.

Proof. A fake block can be generated either by at least $t + 1$ corrupted consensus nodes or by a vulnerability in the underlying threshold signature scheme [35]. In either case, it contradicts with the threshold assumption or the security of the underlying signature scheme. Namely, if less than t nodes are corrupted fake blocks (or block headers) cannot be generated and propagated due to the underlying threshold signature scheme.

□

Property (Revocation). The revocation process is completely conducted on KORGAN and the dependency to CAs is reduced since not only the CAs but also the Certificate Owners can revoke their certificates using the private key of the certificates. Also, the revocation status of all TLS certificates can be tracked and verified transparently. The security analysis of the revocation process of KORGAN is similar to CertLedger, which is discussed in Section 3.5.

Remark. In our proposal, we require a dynamic threshold signature scheme to keep the verification key unchanged in the TLS clients, and preserve the same security level of PBFT independently from adding and removing a consensus node. It is clear that the change in the number of consensus nodes does not affect the faulty assumption of PBFT which does not exceed 33%. Since the nodes in PBFT are stateless, once a node is removed (which cannot be a part of the consensus mechanism anymore) it will not be able to have an impact on the consensus.

However, in our proposal, it is probable that removed nodes can become active and colluding adversaries (considering dynamic adversaries instead of static adversaries). Since each node possesses a partial private key share SK_i , t compromised

nodes (with possibly colluding removed ones) still will be able to use their SK_i s to generate a fake block by creating a valid signature without running a consensus protocol. Therefore, the acceptable level of the threshold t could be changed dynamically. To make the system α -tolerant where $\alpha < t$, we accept to remove α nodes without changing the (PK, SK) key pair. However, if more than α nodes are removed, (PK, SK) should be updated and the new PK should be distributed to the TLS clients. We would like to highlight that the removed nodes not necessarily become colluding adversaries but for the sake of security we assume that PK should be changed once α nodes have been removed. The number α can be adjusted according to security considerations, however we do not expect this removal frequently as the nodes are chosen from reputable organizations like universities, large world-wide companies (like Google), IETF, IEEE, and NIST.

4.4. Implementation and Experimental Results

We have implemented our proposal by updating the CA smart contract of Yakubov et al.'s scheme [63] and experimented certificate validation with our scheme. For the experiments, we deployed two smart contracts on private Ethereum network so that we had two CAs in the trust chain. For generation and verification of the state Merkle proofs, we used Eth-proof node-js API [95]. We executed our experiments on a MacBook Pro with Intel Core i7 (3.1 Ghz) CPU, 16 GB of memory and macOS Mojave OS.

Table 4.1 Experimental Results of TLS Handshake in KORGAN

Number of TLS Certificates in the CA Smart Contract	Header Data Size (bytes)	Account Proof Data Size (bytes)	Storage Proof Data Size (bytes)	TLS Handshake Overhead Total (bytes)
1	535	758	590	1883
100	535	758	1089	2382
1.000	535	758	1557	2850
10.000	535	758	2027	3320

We demonstrate the experimental results for TLS handshake overhead in Table 4.1 and elaborate them as follows. First, *Header* denotes the size of the block header in Ethereum, therefore its size is constant and independent of certificates in the smart contract. Second, *AccountProof* is the proof generated to validate the overall state of the smart contract (i.e., the account) comprising its balance, code, and the stored data. Note that its size grows logarithmically with the number of smart contracts in the blockchain due to its Patricia Tree structure [104]. In our first experiment, there was only one CA smart contract, therefore, the size of the *AccountProof* size is the same independently of number of TLS certificates. Third, *StorageProof* is also generated from Storage Merkle-Patricia Tree which is different for all smart contracts in Ethereum. The root value of this tree is also used while computing the state of the account. The size of the *StorageProof* is $\log_2 n \times c_1 + c_2$ where n is the number certificates in an account, c_1 is a constant calculated by adding hash length forming the Merkle proof with the path length between the nodes, and c_2 is the size of input where its hash is calculated to generate a leaf node in the Merkle tree. Hence, the overall TLS Handshake overhead in our scheme is calculated as

$$|Header| + (|AccountProof| + |StorageProof|) \times n \quad (4.3)$$

where n is the number of CAs in the TLS certificate chain.

Table 4.2 Certificate Validation Durations Based on # of TLS Certificates in KORGAN

Number of TLS Certificates in the CA Smart Contract	Certificate Validation Duration (ms)
1	55,56
100	56,37
1.000	59,24
10.000	60,45

We note that certificate validation network overhead is not given in Yakubov et al.'s scheme. On the other hand, in the conventional PKI, the size of a CRL changes with respect to the number of certificates issued by the CA. Even though there are

CRLs ranging up to 109 MB¹⁸, the CRL size for the median certificate is calculated as 51 KB in [30]. In case of OCSF usage for revocation checking, the average size for an OCSF response is about ~4KB. Moreover, the total network overhead increases with respect to the length of the trust chain.

Table 4.3 Certificate Validation Durations Based on Trustchain Length in KORGAN

Trustchain Length Number CA Smart Contract	Certificate Validation Duration (ms)
1	59,24
2	59,85
3	60,13
5	60,67

Our experimental results in Table 4.2 demonstrate that the number of certificates in a CA smart contract does not significantly effect the certificate validation duration. Furthermore, the number of CAs in the trust chain has also negligible effect to the duration (Table 4.3). However, in Yakubov et al.'s scheme, certificate validation duration increases with respect to the length of the trust chain (see [16, Section 5-A]). It also depends on the network latency between the TLS client, full nodes and the web server. On the other hand, in conventional PKI, if the revocation check of a certificate is performed through OCSF, then the latency only due to the network traffic is around 200 ms [106]. The total duration increases with respect to the number of certificates in the trust chain. Since the size of CRLs are much bigger than OCSF responses, their average downloading latency is also greater then OCSF [107].

¹⁸ Apple hosts 109 MB of CRL at [109].

5. CONCLUSION AND FUTURE WORK

There have been serious security incidents due to corrupted CAs which issued fake but valid TLS certificates. To make CAs more transparent and to verify their operations, public logs and blockchain based PKI models are proposed in recent studies. CT is proposed by Google which has almost half of the browser market share. Google made CT mandatory in Chrome for all issued TLS certificates after April 2018 [96]. However, CT and the other public log based proposals are subject to split-world attacks. Today, the likelihood of performing such an attack is even more with CAs which are also maintaining CT logs. Very recently, the private key for signing the CT log of a well-known Public CA which has a big share in SSL/TLS certificate market is compromised [108]. In this thesis, we propose two new PKI models with certificate transparency based on blockchain, what we called CertLedger and KORGAN.

In CertLedger, we store, validate and revoke TLS certificates and manage Trusted CA certificates on a public blockchain. We make TLS certificate issuance, revocation and Trusted CA management process more transparent to prevent MITM kinds of attacks. We also unify and increase efficiency of certificate validation process for all TLS clients. The underlying blockchain of CertLedger can synchronously updated only upon a consensus of its clients, so the trust to single log operator is eliminated. During TLS handshake, all TLS clients can verify the final state of the certificate which makes split-world attack impossible. The experimental results of implementation of CertLedger demonstrates that, the performance of the TLS handshake and certificate validation through CertLedger is significantly improved compared to the current TLS protocol. A future work would be firstly to implement the full functionality of CertLedger in an existing blockchain framework. Secondly, to experiment the full functionality of CertLedger, some of the major browsers and TLS servers(i.e., web/application servers) could be enhanced to be the light client of the blockchain network. Certificate validation implementation in both browsers and TLS servers has to be modified according to the requirements of CertLedger. With a full implementation of CertLedger, storage requirements of full nodes and CertLedger clients could be experimented and compared with the theoretical analysis. Furthermore, the propagation duration of block headers could be compared with the block time to experiment whether the CertLedger Clients can reach the most up to-

date state of CertLedger. Thirdly, an incentive model could be developed for full nodes of CertLedger due to their important role in both storing, validating, preserving a copy of the blockchain data, and generating Merkle proofs for the TLS servers. Crypto Wallets addons of browsers can be updated to support the incentivization system of CertLedger. One last enhancement proposal for CertLedger could be, in order to eliminate the CertLedger Board Members, introducing an automated mechanism running without human intervention may increase the transparency of the trusted CA management.

In KORGAN, we point out the security and privacy issues of one of the most recent proposals (belonging to Yakubov et al. [63]) and eliminate them by proposing a new PKI architecture. KORGAN is based on permissioned blockchain and utilizing PBFT where the blocks are signed through dynamic threshold signature scheme among consensus nodes. Due to the signed blocks, TLS clients can now easily verify the final states of certificates without requiring to be a peer of the blockchain network. Our experimental results on Ethereum demonstrate that KORGAN does not bring any significant computational and network overhead during certificate validation. Even more, the duration of our certificate validation is less than the previous schemes. Further research work mainly includes modifying KORGAN in such a way that generating CA smart contracts could be restricted to only trustworthy CAs. For this purpose, an international board can be established to audit the CAs and sign the smart contract generation transaction using a threshold signature scheme as well. Secondly PBFT with dynamic threshold signatures could be fully implemented to experiment its performance and compare other BFT based mechanisms.

REFERENCES

- [1] Diffie W., (1998), “The first ten years of public-key cryptography”, Proceedings of the IEEE 76 (5), 560-577.
- [2] International Telecommunication Union (ITU-T), (1988), “X.509: Information Technology-Open Systems Interconnection - The Directory: Public-Key and Attribute Certificate Frameworks”, Standard.
- [3] Rescorla E., (2018), “The Transport Layer Security (TLS) Protocol Version 1.3”. Tech. rep., RFC 8446
- [4] Conti M., Dragoni N., Lesyk V., (2016), “A Survey of Man In The Middle Attacks”, IEEE Communications Surveys Tutorials, 18, 2027–2051
- [5] Falliere N., Murchu L. O., Chien E., (2011), “W32. stuxnet dossier”, White paper, Symantec Corp., Security Response 5 (6).
- [6] Web 1, (2011), <https://blog.comodo.com/other/the-recent-ra-compromise/>, (visited 2020-05-10).
- [7] Web 2, (2012), https://www.researchgate.net/publication/269333601_Black_Tulip_Report_of_the_investigation_into_the_DigiNotar_Certificate_Authority_breach, (visited 2020-05-10).
- [8] Web 3, (2012), <https://www.eweek.com/security/mozilla-asked-to-revoke-trustwave-ca-for-allowing-ssl-eavesdropping>, (visited 2020-05-10).
- [9] Web 4, (2013), <https://securelist.com/turktrust-ca-problems-21/34893/>, (visited 2020-05-10).
- [10] Web 5, (2015), <https://security.googleblog.com/2015/03/maintaining-digital-certificate-security.html>, (visited 2020-05-10).
- [11] Web 6, (2015), https://support.lenovo.com/us/tr/product_security/superfish, (visited 2020-05-10).
- [12] Web 7, (2015), <https://security.googleblog.com/2015/09/improved-digital-certificate-security.html>, (visited 2020-05-10).
- [13] Web 8, (2016), <https://www.bloomberg.com/press-releases/2016-06-13/symantec-to-acquire-blue-coat-and-define-the-future-of-cybersecurity>, (visited 2020-05-10).
- [14] Web 9, (2011), <https://www.imperialviolet.org/2011/05/04/pinning.html>, (visited 2020-05-10).
- [15] Marlinspike M., Perrin T., (2012), “Trust assertions for certificate keys (tack)”. Internet Draft, <https://tools.ietf.org/html/draft-perrin-tls-tack-00>.

- [16] Wendlandt D., Andersen D. G., Perrig A., (2008), “Perspectives: Improving sshstyle host authentication with multi-path probing”, USENIX 2008 Annual Technical Conference, ATC'08, USENIX Association, Berkeley, CA,USA, 321-334.
- [17] Alicherry M., Keromytis A. D., (2009), “Doublecheck: Multi-path verification against man-in-the-middle attacks”, 2009 IEEE Symposium on Computers and Communications, 557-563.
- [18] Web 10, (2020), <https://www.eff.org/observatory>, (visited 2020-05-10).
- [19] Web 11, (2010), <https://www.eff.org/files/ccc2010.pdf>, (visited 2020-05-10).
- [20] Web 12, (2010), <http://patrol.psyced.org>, (visited 2020-05-10).
- [21] Soghoian C., Stamm S., (2012), “Certified Lies: Detecting and Defeating Government Interception Attacks against SSL”, Springer Berlin Heidelberg, 250-259.
- [22] Rivest R. L., (1998), “Can we eliminate certificate revocation lists?”, Springer Berlin Heidelberg, Berlin, Heidelberg, 178-183.
- [23] Web 13, (2012), <https://www.imperialviolet.org/2012/02/05/crlsets.html>, (visited 2020-05-10).
- [24] Laurie B., Langley A., Kasper E., (2013), “Certificate Transparency”, RFC 6962.
- [25] Mazieres D., Shasha D., (2002), “Building secure file systems out of byzantine storage”, Proceedings of the twenty-first annual symposium on Principles of distributed computing, ACM, 108-117.
- [26] Chuat L., Szalachowski P., Perrig A., Laurie B., Messeri E., (2015), “Efficient gossip protocols for verifying the consistency of certificate logs”, 2015 IEEE Conference on Communications and Network Security (CNS), 415-423.
- [27] Nordberg L., Gillmor D., Ritter T., (2018), “Gossiping in CT”, <https://tools.ietf.org/html/draft-ietf-trans-gossip-05>.
- [28] Hof B., (2017), “STH Cross Logging”, <https://tools.ietf.org/pdf/draft-hof-trans-cross-00.pdf>.
- [29] Syta E., Tamas I., Visher D., Wolinsky D. I., Jovanovic P., Gasser L., Gailly N., Khoffi I., Ford B., (2016), “Keeping authorities "honest or bust" with decentralized witness cosigning”, IEEE Symposium on Security and Privacy (SP), 526-545.
- [30] Liu Y., Tome W., Zhang L., Choffnes D., Levin D., Maggs B., Mislove A., Schulman A., Wilson C., (2015), “An end-to-end measurement of certificate revocation in the web's PKI”, Proceedings of the 2015 Internet Measurement Conference, ACM, 183-196.

- [31] Web 13, (2013), <https://www.helpnetsecurity.com/2013/09/16/nsa-impersonated-google-in-mitm-attacks/>, (visited 2020-05-10).
- [32] Wood G., (2014), “Ethereum: A secure decentralised generalised transaction ledger”, Ethereum project yellow paper, 1-32.
- [33] Castro M., Liskov B., (2002), “Practical Byzantine Fault Tolerance and Proactive Recovery”, ACM Transactions on Computer Systems, 20, 398–461.
- [34] Web 14, (2020), <https://www.etsi.org>, (visited 2020-05-10).
- [35] Noack A., Spitz S., (2009) “Dynamic Threshold Cryptosystem without Group Manager”, Network Protocols & Algorithms, 1, 108–121.
- [36] Rivest R., Shamir A., Adleman L., (1978), “A method for obtaining digital signatures and public-key cryptosystems”, Commun. ACM, 21 (2), 120-126.
- [37] Miller V. S., (1986), “Use of elliptic curves in cryptography”, Advances in Cryptology | CRYPTO '85 Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 417-426.
- [38] Koblitz N., (1987), “Elliptic curve cryptosystems”, Mathematics of computation (177), 203-209.
- [39] Rogaway P., Shrimpton T., (2004), “Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, secondpreimage resistance, and collision resistance”, International workshop on fast software encryption, Springer Berlin Heidelberg, 371-388.
- [40] Koblitz N., Menezes A. J., (2015), “The random oracle model: a twenty-year retrospective”, Designs, Codes and Cryptography 77 (2), 587-610.
- [41] Handschuh H., (2000), “SHA-0, SHA-1, SHA-2 (Secure Hash Algorithm)”, Cryptologia 24.2 , 1190-1193. d
- [42] Bertoni G., Daemen J., Peeters M., Van Assche G., (2013), “Keccak”, Annual international conference on the theory and applications of cryptographic techniques, Springer, 2013, 313-314.
- [43] Shamir A., (1979), “How to share a secret”, Commun. ACM 22 (11), 612-613.
- [44] Web 15, (2020), <https://www.win.tue.nl/~berry/CryptographicProtocols/LectureNotes.pdf>, (visited 2020-05-10).
- [45] Adams C., Farrell S., Kause T., Mononen T., (2005), “Internet X. 509 public key infrastructure certificate management protocol (CMP)”, Tech. rep., RFC 4210 (Proposed Standard).
- [46] Schaad J., Myers M., Nystrom M., Kaliski B., (2008), “Certificate management over CMS (CMC)”, Tech. rep., RFC 5272.

- [47] Santesson S., Myers M., Ankney R., Malpani A., Galperin S., Adams C., (2013), “X. 509 Internet Public Key Infrastructure Online Certificate Status Protocol-OCSP”, RFC 6960.
- [48] Merkle R. C., (1980), “Protocols for public key cryptosystems”, IEEE Symposium on Security and Privacy, IEEE, 122-122.
- [49] Barnes R, Hoffman-Andrews J, McCarney D, (2019), “Automatic Certificate Management Environment (ACME)”, RFC 8555.
- [50] Lamport L., Shostak R., Pease M., (1982), “The byzantine generals problem”, ACM Trans. Program. Lang. Syst. 4 (3), 382-401.
- [51] Web 16, (2008), <https://bitcoin.org/bitcoin.pdf>, (visited 2020-05-10).
- [52] Kiayias A., Russell A., David B., Oliynykov R., (2017), “Ouroboros: A provably secure proof-of-stake blockchain protocol”, Advances in Cryptology-CRYPTO 2017, Springer International Publishing, 357-388
- [53] Eckersley P., (2012), “Sovereign key cryptography for internet domains”, Internet draft.
- [54] Yu J., Cheval V., Ryan M., (2016), “DTKI: A new formalized PKI with verifiable trusted parties”, The Computer Journal 59 (11), 1695-1713.
- [55] Crosby S. A., Wallach D. S., (2009), “Efficient data structures for tamper-evident logging”, USENIX Security Symposium, 317-334.
- [56] Ryan M. D., (2014), “Enhanced certificate transparency and end-to-end encrypted mail”, Network and Distributed System Security Symposium (NDSS), The Internet Society.
- [57] Hyun-Jin Kim T., Huang L.-S., Perrig A., Jackson C., Gligor V., (2013), “Accountable key infrastructure (AKI): A proposal for a public-key validation infrastructure”, Proceedings of the 22nd international conference on World Wide Web, ACM, 679-690.
- [58] Basin D., Cremers C., Kim T. H.-J., Perrig A., Sasse R., Szalachowski P., (2014), “ARPKI: Attack resilient public-key infrastructure”, Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, 382-393.
- [59] Web 17, (2020), <https://www.namecoin.org/>, (visited 2020-05-10).
- [60] Fromknecht C., Velicanu D., Yakoubov S., (2014), “Certcoin: A namecoin based decentralized authentication system”, Unpublished class project.
- [61] Axon L., Goldsmith M., (2017), “PB-PKI: A Privacy-aware Blockchain-based PKI”, In Vol. 6. SCITEPRESS, 311-318.

- [62] Wang Z., Lin J., Cai Q., Wang Q., Jing J., Zha D., (2018), “Blockchain-based certificate transparency and revocation transparency”, Financial Cryptography and Data Security, Springer International Publishing, 2018.
- [63] Yakubov A., Shbair W. M., Wallbom A., Sanda D., State R., (2018), “A blockchain-based pki management framework”, IEEE/IFIP Network Operations and Management Symposium, 1-6.
- [64] Chen J., Yao S., Yuan Q., He K., Ji S., Du R., (2018), “Certchain: Public and efficient certificate audit based on blockchain for TLS connections”, IEEE INFOCOM 2018-IEEE Conference on Computer Communications, 2060-2068.
- [65] Matsumoto S., Reischuk R. M., (2017), “IKP: Turning a PKI around with decentralized automated incentives”, 2017 IEEE Symposium on Security and Privacy (SP), IEEE, 410-426.
- [66] Web 18, (2010), <https://www.hyperledger.org/>, (visited 2020-05-10).
- [67] Web 19, (2020), <https://docs.neo.org/docs/en-us/basic/whitepaper.html>, (visited 2020-05-10).
- [68] Web 20, (2020), <https://neo.org>, (visited 2020-05-10).
- [69] Web 21, (2020), <https://ont.io/wp/Ontology-technology-white-paper-EN.pdf>, (visited 2020-05-10).
- [70] Schwartz D., Youngs N., Britto A., (2014) , The Ripple Protocol Consensus Algorithm, Ripple Labs Inc, White Paper.
- [71] Web 22, (2020), <https://ripple.com>, (visited 2020-05-10).
- [72] Web 23, (2020), <https://www.cardano.org/en/home/>, (visited 2020-05-10).
- [73] King S., Nadal S., (2012), “PPCOIN: Peer-to-Peer Crypto-Currency with Proof-of Stake”, self-published paper.
- [74] Sasson E. B., Chiesa A., Garman C., Green M., Miers I., Tromer E., Virza M., (2014), “ZeroCash: Decentralized anonymous payments from Bitcoin”, Security and Privacy (SP), 2014 IEEE Symposium on, 459-474
- [75] Web 24, (1996), http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html, (visited 2020-05-10).
- [76] Web 25, (2018), <https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md>, (visited 2020-05-10).
- [77] Boneh D., Gennaro R., Goldfeder S., (2017), “Using Level-1 Homomorphic Encryption to Improve Threshold DSA Signatures for Bitcoin Wallet

- Security”, Progress in Cryptology - LATINCRYPT 2017, Springer International Publishing, 352-377
- [78] Gennaro R., Goldfeder S., (2018), “Fast multiparty threshold ECDSA with fast trustless setup”, Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, 1179-1194.
 - [79] Cooper D., Santesson S., Farrell S., Boeyen S., Housley R., Polk W., (2008), “RFC 5280: Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile”, IETF..
 - [80] Web 26, (2020), <https://www.ieee.org/index.html>, (visited 2020-05-10).
 - [81] Web 27, (2020), <https://www.ietf.org>, (visited 2020-05-10).
 - [82] Web 28, (2020), <https://github.com/ethereum/web3.js/>, (visited 2020-05-10).
 - [83] Web 29, (2018), https://objective-see.com/blog/blog_0x26.html, (visited 2020-05-10).
 - [84] Web 30, (2017), <https://community.rsa.com/community/products/netwitness/blog/2017/11/03/inaudible-subversion-did-your-hi-fi-just-subvert-your-pc>, (visited 2020-05-10).
 - [85] Web 31, (2020), <https://cwe.mitre.org/data/definitions/295.html>, (visited 2020-05-10).
 - [86] Web 32, (2018), https://www.verisign.com/en_US/domain-names/dnib/index.xhtml, (visited 2020-05-10).
 - [87] Web 33, (2017), <https://www.wired.com/2017/01/half-web-now-encrypted-makes-everyone-safer/>, (visited 2020-05-10).
 - [88] Web 34, (2017), <https://cabforum.org/2017/03/17/ballot-193-825-day-certificate-lifetimes/>, (visited 2020-05-10).
 - [89] Web 35, (2020), https://en.bitcoin.it/wiki/Help:FAQ#Why_do_I_have_to_wait_10_minutes_before_I_can_spend_money_I_received.3F, (visited 2020-05-10) (2019).
 - [90] Web 36, (2020), <https://diskprices.com>, (visited 2020-05-10).
 - [91] Kiayias A., Miller A., Zindros D., (2017), “Non-interactive proofs of proof-of-work.”, IACR Cryptology ePrint Archive 2017 (963), 1-42.
 - [92] Web 37, (2017), https://scalingbitcoin.org/stanford2017/Day1/flyclient_scalingbitcoin.pptx.pdf, (visited 2020-05-10).
 - [93] C. Decker, R. Wattenhofer, Information propagation in the bitcoin network, in: Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on, IEEE, 2013, 1-10.

- [94] Web 38, (2020), <https://www.bouncycastle.org/docs/tlsdocs1.5on/overview-summary.html>, (visited 2020-05-10).
- [95] Web 39, (2020), <https://github.com/zmitton/eth-proof>, (visited 2020-05-10).
- [96] Web 40, (2020), TLS PKI History, <https://www.feistyduck.com/ssl-tls-and-pki-history/>, (visited 2020-05-10) .
- [97] Krawczyk H, (2010), “Cryptographic extraction and key derivation: The HKDF scheme”. Advances in Cryptology – CRYPTO 2010, volume 6223 of Lecture Notes in Computer Science, 631–648.
- [98] Kubilay M. Y., Kiraz M. S., Mantar H. A., (2019), “CertLedger: A New PKI Model with Certificate Transparency Based on Blockchain”, Computer & Security 85, 333-352.
- [99] Web 41, (2020), <https://firstmonday.org/ojs/index.php/fm/article/view/548>.
- [100] Wust K., Gervais A., (2018), “Do you need a blockchain?”, Crypto Valley Conference on Blockchain Technology (CVCBT), IEEE, 45-54.
- [101] Boneh D., Lynn B., Shacham H., (2001), “Short signatures from the weil pairing” , Advances in Cryptology | ASIACRYPT 2001, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, 514-532.
- [102] Cachin C., Kursawe K., Shoup V., (2005), “Random oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography”, Journal of Cryptology 18 (3), 219-246.
- [103] Yin M., Malkhi D., Reiter M. K., Gueta G. G., Abraham I., (2019), “Hotstuff: Bft consensus with linearity and responsiveness”, Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC '19, ACM, 2019, 347-356.
- [104] Web 42, (2020), <https://github.com/ethereum/wiki/wiki/Patricia-Tree>, (visited 2020-05-10).
- [105] Web 43, (2020), <https://developers.libra.org/docs/crates/consensus>, (visited 2020-05-10).
- [106] Web 44, (2020), <https://uptime.netcraft.com/up/reports/performance/OCSP>, (visited 2020-05-10).
- [107] Web 45, (2020), <https://uptime.netcraft.com/up/reports/performance/CRL>, (visited 2020-05-10).
- [108] Web 46, (2020), https://www.feistyduck.com/bulletproof-tls-newsletter/is-sue_65_private_key_of_digicert_certificate_transparency_log_compromised , (visited 2020-05-10).
- [109] Web 47, (2020), <http://crl.apple.com/wwwdrca.crl>, (visited 2020-05-10).

BIOGRAPHY

Murat Yasin Kubilay graduated from the Middle East Technical University (METU) in 1997 and then worked as a software engineer in various companies in Turkey and Germany. He has been appointed head of the software development team for national PKI Project at TUBITAK BILGEM in 2004, and in parallel received his MS degree in Computer Engineering in 2007 from METU. The PKI products developed by his team has been certified with Common Criteria EAL 4+. These PKI products are being widely used in Government Certification Center and in many other state agencies and private sector in Turkey. He has been currently working as a Lead Security Engineer in Deutsche Bank, Germany since November 2018.

APPENDICES

Appendix A: Publications within the PhD Dissertation

M. Y. Kubilay, M. S. Kiraz, H. A. Mantar, CertLedger: A New PKI Model with Certificate Transparency Based on Blockchain, *Computer & Security* 85 (2019) 333-352.

M. Y. Kubilay, M. S. Kiraz, H. A. Mantar, KORGAN: An Efficient PKI Architecture Based on Permissioned-Blockchain by Modifying PBFT Through Dynamic Threshold Signatures, *Computer Journal* (2020), doi: 10.1093/comjnl/bxaa081.

