CertLedger: A New PKI Model with Certificate Transparency Based on Blockchain

Murat Yasin Kubilay[1], Mehmet Sabir Kiraz[2], Haci Ali Mantar[1]

[1] Department of Computer Engineering, Gebze Technical University, Kocaeli, Turkey,
[2] De Montfort University, School of Computer Science and Informatics, Leicester, UK
mkubilay@hotmail.com, mehmet.kiraz@dmu.ac.uk, hamantar@gtu.edu.tr

Abstract. 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 (CT) 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 paper, we propose a new PKI architecture with certificate transparency based on blockchain, what we called *CertLedger*, to eliminate the split-world attacks and to provide certificate/revocation transparency. All TLS certificates' validation, storage, and entire revocation process is conducted in CertLedger as well as Trusted CA certificate management. During a TLS connection, TLS clients get an efficient proof of existence of the certificate directly from its domain owners. Hence, privacy is now perfectly preserved by eliminating the traceability issue via OCSP servers. It also provides a unique, efficient, and trustworthy certificate validation process eliminating the conventional inadequate and incompatible certificate validation processes implemented by different software vendors. TLS clients in CertLedger also do not require to make certificate validation and store the trusted CA certificates anymore. We analyze the security and performance of CertLedger and provide a comparison with the previous proposals. Finally, we implement its protoype on Ethereum to demonstrate experimental results. The results show that the performance of the TLS handshake and certificate validation through CertLedger is significantly improved compared to the current TLS protocol.

Keywords: PKI, SSL/TLS, Certificate Transparency, Certificate validation, Privacy, Blockchain.

1 Introduction

Web based applications such as internet banking, mailing, e-commerce are playing a major role for facilitating our life and became an indispensable part of it. As a de facto standard, SSL/TLS certificates are used to

provide authenticity, integrity, and confidentiality services to these applications. These certificates are issued by CAs which are assumed to be trusted organizations in the conventional PKI systems. 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 [1] 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 [2]. 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 [3]. 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 [4].
- A Turkish CA Turktrust has mistakenly issued subCA certificates instead of TLS certificates in December 2012 [5]. 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 [6]. Later on, it is identified that CNNIC is operated without documented CPS.
- Lenova Superfish has deployed a local CA in its products in 2015 [7]. 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 [8]. Later on Symantec claimed that these certificates were produced for test purposes.
- Symantec purchased Blue Coat in May 2016 [9]. 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 [10,11], crowd sourcing [12,13,14,15,16,17], and pushing revocation information to browsers [18,19] are the initial solutions which are partly implemented but failed due to scalability problems. There are also other recent proposals which are revisited in Section 7. In order to depict the state of the art in the conventional PKI, we describe one of these proposals (Certificate Transparency) which is proposed by Google in the following section.

1.1 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 [20]. *CT* uses append-only Merkle Hash Trees (MHT) for appending and efficiently verifying TLS certificates [21]. In this respect, *CT* introduces *Certificate Log*s ($\mathcal{CL}$), *Monitor*s, and *Auditor*s as new entities. CAs submit the new TLS (pre)certificates to several $\mathcal{CL}$s and each of them generate a cryptographic proof called Signed Certificate Timestamp (SCT). After appending the new certificates to the log, $\mathcal{CL}$ 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 $\mathcal{CL}$'s public key, they may refuse to connect to the service. Domain owners or private companies can act as *Monitor*s and continually inspect certificates of their interest in $\mathcal{CL}$ whether there exist illegitimate certificates. Browsers, or in general TLS clients, act as *Auditor*s. They verify if a $\mathcal{CL}$ is behaving properly and cryptographically consistent.

However, the authors in [22] show that if an adversary can get a fake but valid TLS certificate and can control the $\mathcal{CL}$ 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* [23] which is elaborated as follows.

1. The adversary gets a fake but valid TLS certificate for a domain from the CA.
2. The adversary or the CA submits this certificate to the log which is appended to the fraudulent hash tree.
3. 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.
4. 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.
5. Upon validation of the fake certificate and the SCT, the client connects to either the fake domain or the real domain through a proxy.

*Monitor*s cannot detect this attack since they get and verify the *Consistency Proof*s only from the genuine view of the log. *Auditor*s (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 *Auditor*s 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 [20], 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 *Auditor*s for checking the consistency of the log. Some of these gossiping *Auditor*s 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.

1.2 Ongoing Security Issues

In the existing public log based proposals (described in Section 7), 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 [22]. 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 [23,24,25,26]. 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.

Certificate revocation and validation processes have major problems in today's PKI as also described in [27]. Namely, for the revocation process, certificate owners have to rely on CAs which have the full responsibility to revoke the certificates and give 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 [27]. More importantly, checking the revocation status of a certificate using OCSP also causes privacy concerns [27]. Moreover, incompatible and inadequate implementations of certificate validation and revocation behavior within browsers are also error prone [27].

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 [7,28].

1.3 Our Contributions

In this paper, we propose a new and efficient PKI architecture, what we called *CertLedger*, to make TLS certificates and their revocation status transparent while eliminating the above-mentioned issues. *CertLedger* uses a blockchain based public log to validate, store, and revoke TLS certificates. In summary, we make the following contributions in *CertLedger*:

- Resistance to split-world 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 [29]. 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.

1.4 Roadmap

In Section 2, we describe how the afore-mentioned problems of PKI can be solved by using a public blockchain. In Section 3, we present our proposal *CertLedger*, which is a new PKI model with certificate transparency based on blockchain. We analyse *CertLedger* in terms of security and performance in Section 4 and 5, respectively. In Section 6, we discuss our prototype implementation details and evaluate its results. In Section 7 and 8, we describe the existing proposals and compare them with *CertLedger*. Finally, we conclude the paper with the future work in Section 9.

2 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

$$trx := (M, Signature), M := (PKSender, receiver, data),$$

$$Signature := Sign_{SKSender}(H(M))) \text{ where}$$

$PKSender$ 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 $H(\cdot)$ 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) [30] (which is utilized in HyperLedger Fabric [31]), Delegated BFT [32] (which is utilized in Neo [33], and an improved version VBFT is utilized in Ontology [34]), Proof-of-Work (PoW) (which is utilized in Bitcoin [35], Ethereum [29]), Ripple [36] (which is utilized in Ripple [37]), and Proof-of-Stake (PoS) [38] (which is utilized in Cardano [39], PeerCoin [40]). From now on, for simplicity, we refer to "blockchain" as blockchain network. In general there are also three types of Blockchain: 1) Permissionless (e.g., Bitcoin [35], Ethereum [29], ZCash [41]), 2) Public Permissioned Blockchain (e.g., Ripple [36]), 3) 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 1) Split-world attack, 2) Certificate revocation and validation problems, and 3) Trusted certificate/key store management problems.

First, as described in Section 7, existing proposals try to solve the transparency issue in PKI by introducing (one or more) public logs [20,54,55,56]. 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 [20], "*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 [27], 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 [27] 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.

2.1 Blockchain Characteristics for PKI

Using the decision-sequence of Wüst and Gervais [42], 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.

1. *Do you need to store state?* TLS certificates are being updated continually due to expiration or revocation. The state of the certificates have to be stored and updated whenever necessary.

2. *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 *writer*s will lead to a more reliable and robust log.
3. *Can you use an always online Trusted Third Party (TTP)?* As described in Section 1, 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.
4. *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.
5. *Are all writers trusted?* Even though, all the *writers* are seemingly trusted, some of them can be controlled by a strong adversary.
6. *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. First, it must comprise smart contract infrastructure to implement the required rules for validating the state transitions [43]. Second, 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. Third, 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. Fourth, blockchain architecture should enable TLS clients to verify the final state of the TLS certificates efficiently[3]. Fifth, 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.,

[3] 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. [44]

PBFT [30], DBFT/VBFT [32,34])[4]. Neo [33], Ontology [34], and EOS [45] are some of the candidate blockchain architectures fulfilling these requirements. For 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., $(n+1, 3n)$ [46,47] (a consensus outcome (i.e., new block) can now be verified by anyone without requiring an additional consensus algorithm).

3 CertLedger

CertLedger is a PKI architecture 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 complaint to international standard certificate profiles (e.g., RFC 5280 [48]) 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 [48]. 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".

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

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*.

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 Object*s and to give permission to a state object for transfering 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.

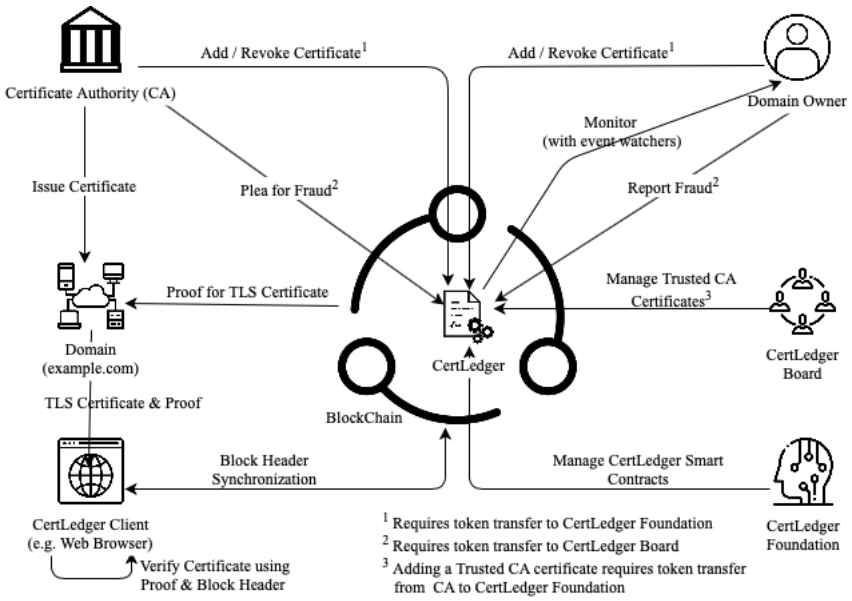

Fig. 1: CertLedger Architecture

3.1 Entities

We have three types of entities in our model: 1) CertLedger Entities (*CertLedger Board*, *CertLedger Foundation*, *CertLedger Clients (TLS Clients)*)

2) External Entities (*Certificate Authorities (CA)*, *Domain Owner*s) 3) Underlying Blockchain Entities (*Miner*s and *Full Node*s).

CAs have basically four different tasks: 1) Check the identity of the *Domain Owner* for his TLS certificate request. 2) Issue a TLS certificate upon successful identity verification. 3) Optionally, create a transaction to append the issued certificate to *CertLedger*. 4) Optionally, create a transaction to change the revocation status of the certificate in *CertLedger*. Therefore, in our model, *CA*s do not issue Certificate Revocation List (CRL) and give OCSP services anymore.

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) [46,47]. More concretely, let $(PKBoard, (SKBoard_1, \cdots, SKBoard_\ell))$ 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.

$$SKBoard := Construct(SKBoard_1, \cdots, SKBoard_\ell)) \tag{1}$$

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 [49], ISO [50] and IETF [51]. 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 [52], ISO [50], and IETF [51]) can be *CertLedger Board* members, but any organization in the world fulling the requirements can be in principle a board member.

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

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.

Miners are *writer*s 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.

Full Nodes store, validate, and propagate all the blocks and generate (Merkle) proofs for the *CertLedger Client*s to verify the final state of the *state object*s.

Domain Owners offer secure services to the *CertLedger Client*s through protocols such as https, imaps, and sips. They have the following tasks: 1) Make TLS certificate request to *CA*s for their domain. 2) Optionally, create a transaction to append the received TLS certificate to *CertLedger*. 3) Monitor their up-to-date TLS certificate in *CertLedger* (e.g., using event listeners on Ontology). 4) In case of compromise detection, immediately create a transaction to revoke their TLS certificate and create another transaction to report the fraud.

3.2 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 *Miner*s are honest with respect to the underlying consensus algorithm for the agreement on the upto date state of the blockchain. Similarly, full nodes are also assumed to be malicious which are allowed to generate fake blocks and proofs. *CertLedger Client*s are also assumed to be malicious. Finally, at most t out of ℓ *CertLedger Board Member*s are assumed to be corrupted. Trivially, we assume also that the underlying

cryptographic primitives of the blockchain architecture of *CertLedger* are secure.

3.3 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 2.

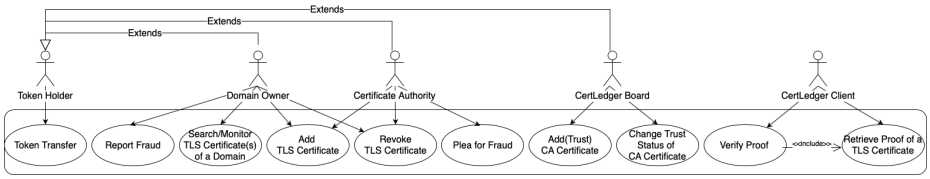

Fig. 2: Functionalities of CertLedger

Algorithm 1 Generating a Transaction for adding a Trusted CA Certificate through Board Members

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

Adding a new Trusted CA Certificate. *CertLedger Board* adds a CA certificate to the *Trusted CAs State Object* as follows (Algorithm 1 and 2): 1) 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 transfering token from its *Account State Object* to the *CertLedger Board*'s *Account State Object*. 2) Upon application of the CA, *CertLedger Board* audits the CA and verifies whether it complies with the Trusted CA standards. 3) 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 [46,47]. 4) 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. 5) Upon successful validation of the CA certificate it is added to the *Trusted CAs State Object* as a trusted certificate. 6) *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*.

Algorithm 2 Adding a Trusted CA Certificate Transaction through Trusted CAs State Object

Input: $trx, certCAList := \{(SKI_j, certCA_j, ts_j) : 1 \leq j \leq m\}, PKBoard$
// 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 $(certCA[SKI], certCA, \text{“}trusted\text{''}) \in certCAList$
if $trx.M.receiver = addr_{TrustedCASO}$ **then**
  if $verify(trx.signature, PKBoard) = true$ **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*
      $certCA[keyUsage] = \text{“}CertificateSigning\text{''}$ *AND*
      $verify\ (certCA[signature],\ certCA[PK]) = true$ // $certCA := trx.M.data$, $certCA[PK]$ denotes the public key of $certCA$
    then
      add $(certCA[SKI], certCA, \text{“}trusted\text{''})$ to $certCAList$
    end
  end
end

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: 1) *CertLedger Board Member*s generate a transaction which is signed in a threshold

fashion to change the status of the CA certificate in the *Trusted CAs State Object*. 2) 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*. 3) The state of the CA certificate is set as "untrusted" in the *Trusted CAs State Object*. 4) *Trusted CAs State Object* triggers *Domain State Object* to change the state of all TLS certificates issued from this CA as "revoked".

Algorithm 3 Adding a TLS Certificate through DomainStateObject

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 $((certTLS[subjectAlternativeName], certTLS, \text{“}valid\text{”}) \in certListURI$ AND $(H(certTLS), certTLS, \text{“}valid\text{”}) \in certListHash)$

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*
      $certTLS[keyUsage] = \text{“}digitalSignature\text{”}$ *AND*
      $certTLS[extendedKeyUsage] = \text{“}serverAuthentication\text{”}$ *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 $(certTLS[subjectAlternativeName], certTLS, \text{“}valid\text{”})$ to $certListURI$
      add $(H(certTLS), certTLS, \text{“}valid\text{”})$ to $certListHash$
    end
  end
end

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 overlap-

ping 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 (Algorithm 3): 1) *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*. 2) 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*. 3) *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*. 4) *Domain State Object* triggers an event notification after addition of the new certificate.

Revocation of a TLS Certificate. A TLS certificate is revoked in *CertLedger* as follows (Algorithm 4): 1) A valid revocation request can only be generated for non-expired and valid TLS certificates. In our model, *Domain Owner*s and *CA*s 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

$$trx.M.data := (revocationMessage, Sign_{SKCertTLS}(H(revocationMessage)))$$

where $revocationMessage := (certTLS, \text{“}revoked\text{”})$. 2) 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*. 3) *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*. 4) *Domain State Object* triggers an event notification after revocation of the certificate.

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: 1)

Algorithm 4 Revoking a TLS Certificate through DomainStateObject

Input: trx, $certCAList$, $certListURI$, $certListHash$ // $trx.M.data$:= ($certTLSToBeRevoked$, $revocationMsg$, $signature$) where $revocationMsg$:= "$revoked$" and $signature$:= $Sign_{SKCertTLS}(H(certTLSToBeRevoked, revocationMsg))$

Output: $\perp$ OR $((certTLS[subjectAlternativeName], certTLS, \text{"}revoked\text{"}) \in certListURI$ AND $(H(certTLS), certTLS, \text{"}revoked\text{"}) \in certListHash)$

// 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 $trx.M.data.revocationMsg$ ="$revoked$" // the domain owner is requesting the revocation **then**

      Change rs of $certTLS$ in $certListURI$ & $certListHash$ as "$revoked$"

    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 $trx.M.data.revocationMsg$ ="$revoked$" // the issuing CA is requesting the revocation **then**

      Change rs of $certTLS$ in $certListURI$ & $certListHash$ as "$revoked$"

    end

  end

end

An entity creates a transaction comprising the amount of the token to be transferred and the destination *Account State Object* address. 2) He signs the transaction with the private key of his *Account State Object*. 3) The transaction triggers the *CertLedger Token State Object*. 4) 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.

Secure Communication through TLS. We assume that most of the *CertLedger Client*s 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). The clients and the TLS server might have different upto date blocks due to network latency. Therefore, a client needs to tolerate upto certain number of old block and accept it as valid. More concretely, assume that the client has nth and the TLS server has $(n - i)$th blocks. If the client tolerates upto

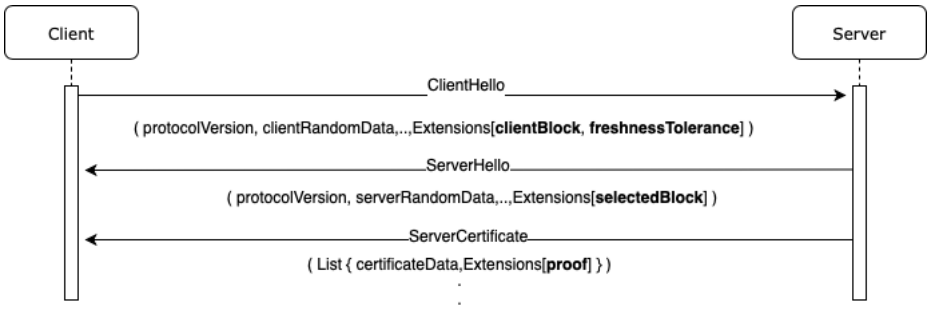

Fig. 3: TLS Handshake Extensions for CertLedger

$-clientBlock$: the extension which holds the block number of the client's latest block header.
$-freshnessTolerance$: the extension which holds the number of old blocks where the client can tolerate for proof generation.
$-selectedBlock$: the extension which holds the block number selected by the server to be used in proof generation. Note that,
$clientBlock.value - freshnessTolerance.value < selectedBlock.value \leq clientBlock.value$
$-proof$: the extension which holds Merkle proof for the TLS certificate generated from the State Tree.

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 Client*s do not require to make further validation of the TLS certificate, since this process is already completed while appending certificate to the blockchain.

Fraud Management

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 [53] for Ethereum. Therefore, in *CertLedger*, monitoring process is effortless and the attack duration can be minimized due to promptness.

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: 1) *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. 2) 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*. 3) 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. 4) If the contract code succeeds, the fraud report is added to the *Fraud Report State Object*. 5) 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*. 6) *Fraud Report State Object* triggers an event notification after adding the new fraud report.

Pleading against a Fraud. CertLedger Board and the *Trusted CA*s 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: 1) 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*. 2) Generates a transaction comprising the hash of these documents signed with its CA certificate. 3) Signs the transaction with the private key of his *Account State Object* and triggers the smart contract code of the *Fraud Report State Object*. 4) 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*. 5) 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*. 6) *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".

4 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 is given in Table 1.

Theorem 1. *Assume that CertLedger is deployed on a blockchain which utilizes PBFT as the consensus algorithm (i.e., at most* n *of 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 [46,47]. 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* $3n$ *(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 *CertLedger* or the status of a Trusted CA certificate can be unfairly changed as 'untrusted', if more than t board members are corrupted[5]. This contradicts with the underlying (t, ℓ) threshold signature scheme [46,47]. 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.

Table 1: Security comparison of Log Based Approaches to Certificate Management

	CT [20]	**AKI** [54]	**ARPKI** [55]	**DTKI** [56]	[57]	[58]	**CertChain** [59]	**CertLedger**
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]	Yes	No	Yes
Eliminates trusted key management	No	No	No	No	No	Yes[c]	No	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]	Yes	No	Yes

[a] Relies on CAs and the revocation process is not transparent
[b] Root certificate validation is still necessary
[c] Eliminates trusted key management in TLS clients, but does not propose a solution for managing Trusted CA certificates in the blockchain
[d] Do not make monitoring due to assuming all master certificates are genuine
[e] Blockchain architecture is inadequate

[5] 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 Member*s.

Fraudulence and plea reports about the Trusted CAs are also managed in *CertLedger*. Any *Domain Owner* who has detected a fake but valid TLS certificate, can put all the fraudulence proofs to *CertLedger*. The accused Trusted CA can also put his plea transactions to *CertLedger* as well. Upon inspection of the proofs, *CertLedger Board Member*s 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 revocation process if Domain Owners cannot use the private keys of their certificates. Hence, it reduces dependency to* CA*s 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:

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 methods [6].

[6] As mentioned in [27], it is difficult to measure revocation reasons, since *Domain Owner*s 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.

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 are completely preserved.*

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 internet browsers. However, management of Trusted CA certificates is a burden and one of the main source of security compromise [60,61]. *CertLedger Client*s do not need local Certificate/Key Stores anymore, since Trusted CA certificates are also managed in *CertLedger*. The trustworthiness of these certificates are 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 [62]. 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 differences in certificate validation implementation. *CertLedger* eliminates this issue by validating TLS certificates prior to appending to the ledger according to the international standards stated in Section 3. 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 Client*s are fully preserved.

5 Performance of CertLedger

5.1 Storage

According to Verisign domain name industry brief, there are about 3.3×10^8 registered domain names at the end of 2017 [63], and approximately half of them are using TLS certificates [64]. 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 CAB Forum Ballot [65].

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 [66,38]). While making the calculations, for illustration purposes we select the target block time as 1 minute because 1) the required storage capacity of the *CertLedger Client*s will increase with shorter block times where some of them have storage constraints and cannot be upgraded easily. 2) 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 *CertLedger*[7]. 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

$$BlockTotalTrx := TotalCert/AnnualGenBlock$$

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

$$|Block| := |Header| + |trx| \times BlockTotalTrx.$$

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

$$|trx| := |Msg| + |Sgn|$$

where $|Msg| := |PKSender| + |Receiver| + |Data|$.

If we assume that the creation of TLS certificates with 1 year average lifetime is dispersed homogeneously through out 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

$$|Blockchain| := AnnualGenBlock \times |Block|$$

Using the EC certificates, the average size of trx is

$$660bytes(\approx 64byte + 20byte + 2^9byte + 64byte),$$

and the average size of $Block$ is

$$128KB(\approx 508^8byte + 660byte \times (1.65 \times 10^8/5\,256\,000).$$

Consequently, the annual average size of $Blockchain$ is 512 GB ($5\,256\,000 \times 128KB$). Considering the fact that the price of a 1 GB disk storage is approximately 0.02 USD [67], the cost of storing *CertLedger* transactions for a full node is only about $10,24.

[7] 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.

[8] Block header size of Ethereum [29].

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 [35], the required disk space to store block headers in Bitcoin for 10 minutes block interval time is 4.2 MB ($6 \times 24 \times 365 \times 80$ byte) 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 a two years of period, *CertLedger Client*s will require approximately 256 MB of disk space for all TLS protected domains[9]. However, this is the maximum amount of data to be stored by the *CertLedger Client*s, we expect that it is sufficient to store the block headers for a much shorter period of time according to their security requirements.

5.2 TLS Communication

The size of the data stored in TLS extension (for Merkle Proof) is $log\ n \times$ *state object* size $+ log\ m \times size(TLSCertificate)$ where n is the number of *state object*s in the state tree and m is the number of certificates in the data storage tree[10]. We assume that there is a single *Account State Object* for each TLS protected domain, and therefore, the number of *state object*s is equal to the TLS protected domains [64]. Consequently, the total communication overhead of a TLS handshake is $(log_2\ 1.65 \times 10^8) \times 16$ byte[11]$+ (log_2\ 1.65 \times 10^8) \times 2^9$ byte ≈ 14 KB.

5.3 Scalability of the Blockchain

According to [70], 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

[9] There are also other studies proposed for efficient light node clients in recent studies such as NIPoPoWs by IOHK [68] and FlyClient by Luu et al. [69]. 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.

[10] 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.

[11] Approximate size of an Account State Object in Ethereum.

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 in the underlying blockchain network of *CertLedger*. Hence, most of the peers of the blockchain network of *CertLedger* receive the new block header within block time. Consequently, *CertLedger Client*s can also reach the most upto-date state of *CertLedger* blockchain within block time.

Table 2: TLS Handshake Experimental 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

6 Implementation and Experimental Results

We developed a prototype[12] of *CertLedger* on a private Ethereum network with a limited set of functionality. In this prototype, our smart contract stores TLS certificates and their revocation status but does not validate them due to the constraints in certificate parsing [13]. 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 [71]. To get and validate Merkle state proof for the TLS certificates we used Eth-proof node-js API [72]. We used a MacBook Pro with Intel Core i5(2,3 GHz) CPU, 16 GB memory and macOS Mojave OS.

In this prototype, we mainly focused on evaluating TLS performance in terms of communication overhead and certificate validation duration. In experimental results (Table 2), we observe that the size of the Merkle proof with respect to the number of certificates in the blokchain is compatible with our theoretical analysis in Section 5.2. Proof verification durations are very close to each other and do not change significantly

[12] The prototype is available on https://github.com/certledgerpki/certledger/.

[13] Note that validating a certificate is trivial and does not bring additional computational complexity.

with the increasing number of certificates. In the conventional PKI, the CRL size for median certificate is 51 KB [27] 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 [73] without validating it.

7 Related Work

In this section, we briefly describe the previous attempts for solving the issues described in Section 1 in a chronological order, and point out their potential weaknesses.

7.1 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 [74]. 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 ($\mathcal{TS}$) along with their domain name. $\mathcal{TS}$ 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 $\mathcal{TS}$, they refuse to make the connection to the service. However, $\mathcal{TS}$ does not return any verifiable proof to domains whether the sovereign key is appended to the log [56]. Furthermore, another strong assumption of *SK* is that clients have to trust not only the $\mathcal{TS}$ but also their Mirrors [56].

7.2 Certificate Issuance and Revocation Transparency (CIRT)

CIRT focuses on extending *CT* to provide additional proofs so that CAs can show their honest behavior [75]. 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*.

7.3 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 [54]. All entities either monitor or report the operations performed by other entities in order to distribute the accountability among the participating entities. *Certification Agency (CA)*, *Integrity Log Server (ILS) Operators (ILSO)*, and *Validator*s are the new entities introduced by *AKI*. *CA*s still issue certificates to domains but they are not the absolute authority for certificate management anymore. *ILSO*s store not only the certificates, but also their registration, update, and revocation information lexicographically according to domain names. *Validator*s monitor *ILS* operations and check whether they operate as expected. Domain owners can select the *CA*s and the *ILSO*s 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 *ILS*s. Domains send their certificates and the verification information received from the *ILS*s 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, *ILS*s, and *Validator*s) 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 *CA*s and the *ILS*s but also the *Validator*s in *AKI* which is a burden for them in terms of trusted entity management.

7.4 Attack Resilient PKI (ARPKI)

ARPKI [55] 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 *CA*s

and one *ILS* are required. For the initial registration process, a domain owner selects the trusted entities, at minimum two *CA*s and one *ILS* (i.e., CA_1, CA_2, and ILS_1). The domain owner designates one of the *CA*s (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 *ILS*s. 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 *ILS*s may lead to a single point of failure in *ARPKI*.

7.5 NameCoin, CertCoin, and PB-PKI

NameCoin [76], which is a fork of Bitcoin [35], is designed to act as a decentralized DNS for ".bit" adresses. 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 [77] 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 [78] 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 ubitiquitous 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.

7.6 Distributed Transparent Key Infrastructure (DTKI)

DTKI [56] 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 ($\mathcal{CLM}$) and Mapping Log Maintainer ($\mathcal{MLM}$) are the two new entities introduced by *DTKI*. $\mathcal{CLM}$s keep all valid, revoked and expired certificates for a set of domains and provide proofs for existence or absence of them. $\mathcal{MLM}$ maintains the association between a set of domain names and the $\mathcal{CLM}$s which are maintaining the logs for them. Mirrors maintain a full copy of the data stored by both the $\mathcal{CLM}$s and the $\mathcal{MLM}$. *CA*s 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 $\mathcal{CLM}$. Users or in particular browsers first query the $\mathcal{MLM}$ in order to find the correct $\mathcal{CLM}$ for a specific domain. To make a connection decision, first the proofs received from the $\mathcal{MLM}$ is verified, then $\mathcal{CLM}$ 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 *CA*s 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 $\mathcal{CLM}$ are both compromised, *DTKI* would not be able to prevent fake Master and TLS certificate issuance. From this perspective, the adversary who is controlling $\mathcal{CLM}$ 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.

7.7 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) [57]. Briefly, in this scheme, web servers publish their TLS certificates to the blockchain using their *publishing key pair*s 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. 1) It has an unreliable basis for providing the trustworthiness of *publishing key pair*s. 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. 2) 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. 3) 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. 4) The proposal is also inefficient in terms of storage costs due to following design considerations. a) A TLS certificate is added to the blockchain periodically during its lifetime, b) 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), c) *Publishing key pair*s are added to the blockchain periodically d) It has large size headers which comprise DNS names existing in the transactions of the block.

7.8 A Blockchain-Based PKI Management Framework

In [58] 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 are stored in the blockchain. This is also rather critical since a detection capability is one of the most crucial requirement 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.

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

CertChain [59] 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 [38], 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.

7.10 IKP: Turning a PKI Around with Decentralized Automated Incentives

The authors in [79] 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 incentivise 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 [79, 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 1.2.

8 Comparison

We now compare *CertLedger* with the other proposals according to following criterias which is tabulated in Table 3.

Log Proofs. *CertLedger* provides proof for the existence and revocation status for all the TLS certificates as *DTKI*, *CertChain*, and [58]. *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. [57] 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.

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 [57], 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 [58], 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. Afterwords, they have to call a bookkeper to check the certificate's existence and its revocation status in the blockchain. *CertLedger Client*s receive approximately 8 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: Comparison of Log Based Approaches to Certificate Management

	CT [20]	**AKI**[54]	**ARPKI**[55]	**DTKI**[56]	[57]	[58]	**CertChain**[59]	**CertLedger**
External Dependency During TLS Handshake	Yes	Yes	Yes[a]	Yes	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 validation through Smart contract or Web service	Certificate signature validation locally, Certificate validation through Bookkeepers	Data transmission overhead and verification of the Merkle audit proof
Existence of Logs with Different Contents	Yes	Yes	Yes	No	No	Yes	No	No
Necessity of External Auditing	Yes	Yes	Yes	Yes	No	No	No	No
Necessity of External Monitoring	Yes	Yes	Yes	No [b]	Yes	Yes	Yes	Yes[c]
Does TLS Clients Require to store Trusted Keys or Certificates of Logs ?	Yes	Yes	Yes	Yes	Yes	No	Yes	No
TLS Clients Requires to Store a copy of the Log	No	No	No	No	Partly (block headers)	No [d]	No *d*	Partly (few block headers)

[a] Clients have to make standard certificate validation which may require external resources
[b] Makes an unrealistic assumption that fake Master certificates cannot be issued, apparently not relevant for strong adversaries
[c] Monitoring is efficient, it is possible to be aware of the TLS certificate state change promptly
[d] TLS clients in [58] and CertChain cannot verify certificate validation responses

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 Client*s do not need a trusted key for the verification of the log proofs. Only this single log have to be monitored.

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.

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 *Validator*s which check the consistency of *ILS*s and whether the TLS certificates are updated according to certificate policies. In *ARPKI*, *Validator*s are optional and their role is distributed to other CAs or *ILS*s. *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. [57] does not require external auditing, however it does not propose an efficient monitoring architecture. *CertChain*, [58] 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.

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 *SCT*s. Although, *AKI* states that installing the trusted CAs and *ILS*s 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 *Validator*s' public keys. In *ARPKI*, TLS clients have to store the trusted CAs and *ILS*s 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 [57] clients continue to make the conventional trusted key/certificate store management. [58] 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 Client*s 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.

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 en-

ables 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 [57] continue to make conventional revocation checking, hence their browsing history can be tracked by the OCSP servers. [58] 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, privacy of the clients are fully preserved since the Merkle state proofs are provided by *Domain Owners*.

9 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 [80]. However, CT and the other proposals are subject to split-world attacks. In this paper, we propose a new PKI model with certificate transparency based on blockchain, what we called *CertLedger*. In *CertLedger* all TLS clients can verify the final state of the log, which makes split-world attack impossible. Moreover, *CertLedger* also provides transparency in certificate revocation and trusted CA management processes. A future work would be implementing the full functionality of *CertLedger* in an existing blockchain framework. Moreover, in order to eliminate the *CertLedger Board Member*s, introducing an automated mechanism running without human intervention may increase the transparency of the trusted CA management.

References

1. Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5(6):29, 2011.
2. Phillip. Comodo ssl affiliate the recent ra compromise, March 2011. `https://blog.comodo.com/other/the-recent-ra-compromise/` (visited 2019-04-08).
3. Diginotar, March 2011. `https://en.wikipedia.org/wiki/DigiNotar` (visited 2019-04-08).

4. Mozilla asked to revoke trustwave CA for allowing ssl eavesdropping, February 2012. https://www.eweek.com/security/mozilla-asked-to-revoke-trustwave-ca-for-allowing-ssl-eavesdropping (visited 2019-04-08).
5. Turktrust CA problems, January 2013. https://securelist.com/turktrust-ca-problems-21/34893/ (visited 2019-04-08).
6. Maintaining digital certificate security, March 2015. https://security.googleblog.com/2015/03/maintaining-digital-certificate-security.html (visited 2019-04-08).
7. Superfish vulnerability, March 2015. https://support.lenovo.com/tr/en/product_security/superfish (visited 2019-04-08).
8. Improved digital certificate security, September 2015. https://security.googleblog.com/2015/09/improved-digital-certificate-security.html (visited 2019-04-08).
9. Symantec to acquire blue coat and define the future of cybersecurity, June 2016. https://www.symantec.com/about/newsroom/press-releases/2016/symantec_0612_01 (visited 2019-04-08).
10. A. Langley. Public-key pinning. imperialviolet, May 2011.
11. M. Marlinspike and T. Perrin. Trust assertions for certificate keys (tack). internet draft, 2012. https://tools.ietf.org/html/draft-perrin-tls-tack-00.
12. Dan Wendlandt, David G. Andersen, and Adrian Perrig. Perspectives: Improving ssh-style host authentication with multi-path probing. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 321–334, Berkeley, CA, USA, 2008. USENIX Association.
13. M. Alicherry and A. D. Keromytis. Doublecheck: Multi-path verification against man-in-the-middle attacks. In *2009 IEEE Symposium on Computers and Communications*, pages 557–563, July 2009.
14. EFFSSL. The EFF SSL observatory. https://www.eff.org/observatory (visited 2019-04-08).
15. P. Eckersley and J. Burns. Is the SSLiverse a safe place? chaos communication congress., 2010. https://www.eff.org/files/ccc2010.pdf (visited 2019-04-08).
16. Certificate patrol. http://patrol.psyced.org (visited 2019-04-08).
17. Christopher Soghoian and Sid Stamm. *Certified Lies: Detecting and Defeating Government Interception Attacks against SSL (Short Paper)*, pages 250–259. Springer Berlin Heidelberg, 2012.
18. Ronald L. Rivest. *Can we eliminate certificate revocation lists?*, pages 178–183. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.
19. A. Langley. Revocation checking and Chrome's CRL. https://www.imperialviolet.org/2012/02/05/crlsets.html (visited 2019-04-08).
20. B. Laurie, A. Langley, and E. Kasper. Certificate Transparency, RFC6962, 2013.
21. Scott A Crosby and Dan S Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, pages 317–334, 2009.
22. David Mazieres and Dennis Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 108–117. ACM, 2002.
23. L. Chuat, P. Szalachowski, A. Perrig, B. Laurie, and E. Messeri. Efficient gossip protocols for verifying the consistency of certificate logs. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 415–423, Sep. 2015.
24. L. Nordberg, D. Gillmor, and T. Ritter. Gossiping in CT, 2018. https://tools.ietf.org/html/draft-ietf-trans-gossip-05,.

25. B. Hof. STH Cross Logging, 2017. https://tools.ietf.org/pdf/draft-hof-trans-cross-00.pdf,.
26. E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford. Keeping authorities "honest or bust" with decentralized witness cosigning. In *IEEE Symposium on Security and Privacy (SP)*, pages 526–545, May 2016.
27. Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. An end-to-end measurement of certificate revocation in the web's PKI. In *Proceedings of the 2015 Internet Measurement Conference*, pages 183–196. ACM, 2015.
28. NSA impersonated Google in MitM attacks. https://www.helpnetsecurity.com/2013/09/16/nsa-impersonated-google-in-mitm-attacks/ (visited 2019-04-08).
29. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.
30. Miguel Castro, Barbara Liskov, et al. Practical Byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
31. Hyperledger project. https://www.hyperledger.org/ (visited 2019-04-08).
32. NEO white paper. https://docs.neo.org/en-us/ (visited 2019-04-08).
33. NEO - An Open Network For Smart Economy. https://neo.org. (visited 2019-04-08).
34. Ontology technology whitepaper. https://ont.io/wp/Ontology-technology-white-paper-EN.pdf (visited 2019-04-08).
35. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
36. David Schwartz, Noah Youngs, Arthur Britto, et al. The Ripple Protocol Consensus Algorithm. *Ripple Labs Inc White Paper*, 5, 2014.
37. Ripple. https://ripple.com (visited 2019-04-08).
38. Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Advances in Cryptology – CRYPTO 2017*, pages 357–388. Springer International Publishing, 2017.
39. Cardano. https://www.cardano.org/en/home/ (visited 2019-04-08).
40. Sunny King and Scott Nadal. PPCOIN: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19, 2012.
41. Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. ZeroCash: Decentralized anonymous payments from Bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 459–474. IEEE, 2014.
42. Karl Wüst and Arthur Gervais. Do you need a blockchain? *IACR Cryptology ePrint Archive*, 2017:375, 2017.
43. Smart contracts: Building blocks for digital markets. http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html (visited 2019-04-08).
44. Merkle Patricia Tree. https://github.com/ethereum/wiki/wiki/Patricia-Tree (visited 2019-04-08).
45. EOS.IO technical white paper v2. https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md (visited 2019-04-08).
46. S. Goldfeder D. Boneh, R. Gennaro. Using level-1 homomorphic encryption to improve threshold dsa signatures for bitcoin wallet security, 2017. http://www.cs.haifa.ac.il/~orrd/LC17/paper72.pdf[online]Available/.

47. Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1179–1194. ACM, 2018.
48. RFC 5280: Internet X.509 Public Key Infrastructure: Certificate and CRL profile. https://tools.ietf.org/html/rfc5280.
49. Institute of Electrical and Electronics Engineers-IEEE. https://www.ieee.org/index.html.
50. International Organization for Standardization. https://www.iso.org/home.html (visited 2019-04-08).
51. Internet Engineering Task Force. https://www.ietf.org (visited 2019-04-08).
52. European Telecommunications Standards Institute -ETSI. https://www.etsi.org (visited 2019-04-08).
53. Web3.js - ethereum javascript api. https://github.com/ethereum/web3.js// (visited 2019-04-08).
54. Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. Accountable Key Infrastructure (AKI): A Proposal for a Public-Key Validation Infrastructure. In *Proceedings of the 22nd international conference on World Wide Web*, pages 679–690, New York, NY, 2013. ACM.
55. David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: Attack resilient public-key infrastructure. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14.
56. Jiangshan Yu, Vincent Cheval, and Mark Ryan. DTKI: A new formalized PKI with verifiable trusted parties. *The Computer Journal*, 59(11):1695–1713, 2016.
57. Ze Wang, Jingqiang Lin, Quanwei Cai, Qiongxiao Wang, Jiwu Jing, and Daren Zha. Blockchain-based certificate transparency and revocation transparency. In *Financial Cryptography and Data Security*. Springer International Publishing, 2018.
58. A. Yakubov, W. M. Shbair, A. Wallbom, D. Sanda, and R. State. A blockchain-based pki management framework. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–6, April 2018.
59. Jing Chen, Shixiong Yao, Quan Yuan, Kun He, Shouling Ji, and Ruiying Du. Certchain: Public and efficient certificate audit based on blockchain for tls connections. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2060–2068. IEEE, 2018.
60. Patrick Wardle. Ay MaMi. https://objective-see.com/blog/blog_0x26.html (visited 2019-04-08).
61. Symantec. Marketscore proxyserver certificate. https://www.symantec.com/security_response/attacksignatures/detail.jsp?asid=20804 (visited 2019-04-08).
62. CWE-295: Improper certificate validation. https://cwe.mitre.org/data/definitions/295.html (visited 2019-04-08).
63. The Verisign Domain Name Industry Brief Q4 2018. https://www.verisign.com/en_US/domain-names/dnib/index.xhtml (visited 2019-04-08).
64. Half the web is now encrypted. https://www.wired.com/2017/01/half-web-now-encrypted-makes-everyone-safer/ (visited 2019-04-08).
65. Certificate Lifetimes Ballot. https://cabforum.org/2017/03/17/ballot-193-825-day-certificate-lifetimes/ (visited 2019-04-08).
66. Bitcoin wiki. https://en.bitcoin.it/wiki/Help:FAQ#Why_do_I_have_to_wait_10_minutes_before_I_can_spend_money_I_received.3F (visited 2019-04-08).
67. HDD Disk Prices. https://diskprices.com (visited 2019-04-08).

68. Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. Non-interactive proofs of proof-of-work, 2017.
69. Zamani M. Luu L., Bunz B. Flyclient:Super Light Client For CryptoCurrencies. `https://scalingbitcoin.org/stanford2017/Day1/flyclientscalingbitcoin.pptx.pdf` (visited 2019-04-08).
70. Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*, pages 1–10. IEEE, 2013.
71. Bouncy Castle JSSE Provider. `https://www.bouncycastle.org/docs/tlsdocs1.5on/overview-summary.html` (visited 2019-04-08).
72. EthProof. `https://github.com/zmitton/eth-proof` (visited 2019-04-08).
73. NetCraft. OCSP Server Performance in April 2013. `https://news.netcraft.com/archives/2013/05/23/ocsp-server-performance-in-april-2013.html`.
74. P. Eckersley. Sovereign key cryptography for internet domains. Internet draft., 2012.
75. Mark D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *In Network and Distributed System Security Symposium (NDSS) The Internet Society*, 2014.
76. Namecoin. `https://www.namecoin.org/` (visited 2019-04-08).
77. Conner Fromknecht, Dragos Velicanu, and Sophia Yakoubov. Certcoin: A namecoin based decentralized authentication system 6.857 class project. *Unpublished class project*, 2014.
78. LM Axon and Michael Goldsmith. PB-PKI: A privacy-aware blockchain-based PKI. 2016.
79. Stephanos Matsumoto and Raphael M Reischuk. IKP: Turning a PKI around with decentralized automated incentives. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 410–426. IEEE, 2017.
80. TLS PKI History. `https://www.feistyduck.com/ssl-tls-and-pki-history/` (visited 2019-04-08).