Article

Associative Blockchain for Decentralized PKI Transparency

Xavier Boyen [1], Udyani Herath [1], Matthew McKague [1,*] and Douglas Stebila [2]

[1] School of Computer Science, Queensland University of Technology, Brisbane City, QLD 4000, Australia; xb@boyen.org (X.B.); udyani.herath@gmail.com (U.H.)
[2] Department of Combinatorics & Optimization, University of Waterloo, Waterloo, ON N2L3G1, Canada; dstebila@uwateroo.ca
* Correspondence: matthew.mckague@qut.edu.au

Abstract: The conventional public key infrastructure (PKI) model, which powers most of the Internet, suffers from an excess of trust into certificate authorities (CAs), compounded by a lack of transparency which makes it vulnerable to hard-to-detect targeted stealth impersonation attacks. Existing approaches to make certificate issuance more transparent, including ones based on blockchains, are still somewhat centralized. We present decentralized PKI transparency (DPKIT): a decentralized client-based approach to enforcing transparency in certificate issuance and revocation while eliminating single points of failure. DPKIT efficiently leverages an existing blockchain to realize an append-only, distributed associative array, which allows anyone (or their browser) to audit and update the history of all publicly issued certificates and revocations for any domain. Our technical contributions include definitions for append-only associative ledgers, a security model for certificate transparency, and a formal analysis of our DPKIT construction with respect to the same. Intended as a client-side browser extension, DPKIT will be effective at fraud detection and prosecution, even under fledgling user adoption, and with better coverage and privacy than federated observatories, such as Google's or the Electronic Frontier Foundation's.

Keywords: certificate transparency; blockchain; digital certificates

check for updates

Citation: Boyen, X.; Herath, U.; McKague, M.; Stebila, D. Associative Blockchain for Decentralized PKI Transparency. *Cryptography* **2021**, *5*, 14. https://doi.org/10.3390/cryptography5020014

Academic Editor: Rongxing Lu

Received: 21 March 2021
Accepted: 25 May 2021
Published: 28 May 2021

1. Introduction

By far, the predominant approach for securing data in transit over the Internet is based on a hierarchical public-key infrastructure (PKI), powered by the Transport Layer Security (TLS) protocol [1], with the use of X.509 certificates [2], rooted in a few hundred of certificate authorities (CAs), themselves anointed or excluded by a mere handful of dominant operating-system and web-browser vendors. Communication security under this model requires the safe distribution and identification of public keys, allowing users to verify their counterpart or the site they connect to.

Unfortunately, this model introduces many single points of failure, whereby any one of the hundreds of browser-trusted CAs is able to mount a man-in-the-middle impersonation attack against any domain, using their ability to issue valid but unauthorised certificates for the intended target. Without user vigilance, this type of attack can remain undetected for a long time, especially when performed sporadically against specific users, or against users in unfamiliar environments.

These threats are not idle speculation: there have been several high-profile cases of mis-issued certificates being used to spoof legitimate websites, eventually resulting in the eviction of the offending CAs in most popular web browsers. In 2011, a prominent Dutch certificate authority, DigiNotar [3], was hacked, and the attackers managed to issue themselves a valid certificate for the domain `google.com` and its sub-domains, compromising Google's identity retention. Similar cases that happened more recently involved CAs by the Comodo group, Turktrust, Thawte, Trustwave and WoSign [4–6].

Though those cases were popularised, many more may have flown under the radar. As a domain can be bound by several digital certificates, it is challenging for clients to differentiate between legitimate ones, and unauthorised forgeries that portend man-in-the-middle attacks. Another issue with the PKI model is the inefficiency of its revocation mechanism, especially with the recent surge of HTTPS-enabled web sites—"Let's Encrypt" being a particularly prolific CA with over 380 million issued certificates in its three-year existence. Traditional revocation mechanisms, from offline Certificate Revocation Lists [7], to Online Certificate Status Protocol (OCSP) and OCSP Stapling [8,9], are simply not efficient or reliable enough to handle a large scale attack; and while identity-based-encryption approaches may theoretically fare better in that regard [10], they would constitute a big step backward in terms of the decentralisation of trust.

Mitigation strategies against those issues fall into two categories: incompatible PKI redesigns, and compatible PKI add-ons. Our objective in this paper falls in the latter category. We seek to create a client-driven, fully decentralised certificate transparency mechanism, that any interested user can privately run in their own browser, in order to catch and deter CA trust abuse. Compared with prior proposals, ours features a better combination of decentralisation, privacy, and compatibility with existing infrastructure and reluctant participants.

1.1. Our Contribution

We propose decentralised PKI transparency (DPKIT), a fully decentralised approach to "certificate transparency", that seeks to rectify PKI trust issues on a user-driven voluntary basis, without requiring any change to the underlying hierarchical PKI. There are two reasons to stick with PKI: (1) compatibility with existing systems; and (2) the recognition that traditional CAs play a valuable role in vetting and vouching for the identity of domain owners "in the real world".

To achieve this, DPKIT provides a strongly immutable record of all issued and revoked certificates, as seen by the worldwide community of users, that efficiently supports all modern use cases of certificates in actual web protocols (including revocations, multiple certificates per domain, multiple domains per certificates, and so on). The reporting function of DPKIT allows participating users to record any and all valid certificate or revocation that is presented to them, in order to provide a global audit. By design, the audit function of DPKIT also doubles as a verifier-local freshness and revocation checking mechanism, with much greater privacy than the industry-standard OCSP online certificate validation.

Abstractly, DPKIT can be thought of as an associative distributed ledger, resulting from the subornation of an associative array to a secure ledger in the blockchain sense, combined with efficient proofs of membership as in Merkle trees.

Note that our intention is to describe the data structures, algorithms, and security of our scheme only, and we do not provide an implementation. It is also not our intention to consider the security of distributed ledgers, and hence we model them with an ideal functionality which may be replaced with a secure ledger of choice.

1.1.1. Properties and Design Goals

Our purpose is to design a secure decentralised system to provide maximum transparency to an existing public-key infrastructure. Our main design goals are:

- *Transparency*: Provide a publicly auditable system that enables anyone who connects to verify certificates or detect misbehaviour.
- *Multiple certificates*: Allow an entity to register multiple certificates per domain, mapping one identity to multiple public keys.
- *Multiple domain names*: Handle multiple domain names on a single certificate.
- *No single point of failure*: No centralisation or federation of any kind.
- *Scalability*: Remain efficient as more and more identities are recorded.
- *Efficiency*: Have low impact on TLS servers and minimal client storage/processing.
- *Client privacy*: Keep end-user browsing habits maximally private from observers and third parties.
- *Revocation*: Provide a reliable revocation system.

1.1.2. Overview of Our Construction

Our construction is essentially a scheme for indexing and proving membership of certificates and their associated revocations, if applicable. This is done through three data structures, as described below and depicted in Figure 1:

- *Certificate trees:* A set of Merkle binary search trees (described in Section 2.1)—one per subject name—that stores all certificates associated with that subject name. Revocations are stored alongside the associated certificate. Certificates are indexed by the fingerprint and revocations are indexed by the fingerprint of the certificate they revoke. These structures provide short proofs of membership. Since revocations are stored alongside certificates, a proof of membership for a certificate reveals whether a revocation exists in the store.
- *Primary tree:* A Merkle Binary Search tree which stores pointers to certificate trees, indexed by subject names. Note that it is possible to paste together membership proofs from these two trees to prove the membership of a certificate in the overall store.
- A distributed ledger (described in Section 2.2) which is used to store updates to the trees in the form of the new root hash of the primary tree, along with any certificates and revocations that have been added since the last ledger entry.

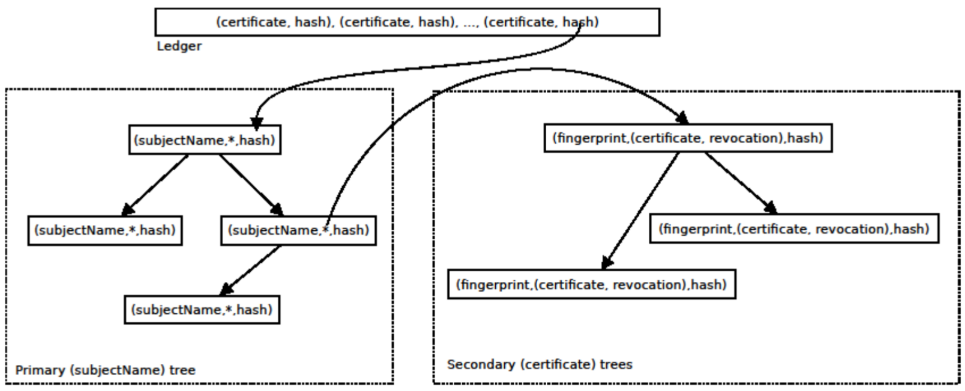
(certificate, hash), (certificate, hash), ..., (certificate, hash)
Ledger
(subjectName,*,hash)
(subjectName,*,hash)
(subjectName,*,hash)
(subjectName,*,hash)
Primary (subjectName) tree
(fingerprint,(certificate, revocation),hash)
(fingerprint,(certificate, revocation),hash)
(fingerprint,(certificate, revocation),hash)
Secondary (certificate) trees

Figure 1. Overview of our construction. The primary tree uses subject names as keys, and each key points to a secondary tree. Each secondary tree stores (certificate, revocation) pairs with certificate fingerprints as keys. The ledger stores a (certificate, hash) for each certificate as they are added, where the hash is the root hash of the primary tree.

The ledger's role is to store the list of certificates and root hashes. In principle, the store could be implemented solely by recording certificates on the ledger, certificates are stored in both the ledger and the trees. However, storing the root hashes on the ledger and maintaining the trees allows for fast searching and short proofs of membership, which can be verified without access to the entire ledger. Given a certificate and a certificate membership proof—obtained from a server—and the most recent entry in the ledger—obtained from a trusted peer—a client can verify that the certificate exists in the store, and determine whether there are any revocations for the certificate. Since tree construction is deterministic, a peer with a copy of the ledger can reconstruct and verify the trees by adding the certificates and revocations from the ledger in order, after which the root hash can be verified.

1.2. Background and Related Work

We first discuss some of the literature that aims to improve PKI security. We split this into three groups: PKI transparency, which aims to improve the security of traditional PKI by logging certificates, blockchain-based PKI schemes, which replace CAs with a decentralised system, and schemes which augment the traditional CA-based PKI with blockchains.

1.2.1. PKI Transparency

The purpose of certificate transparency (CT), is to improve a PKI's transparency in order to catch and publicly expose misdeeds, allowing stakeholders such as domain holders to take action against misbehaving CAs.

Classic log-based PKI extensions allow domain owners to learn when fraudulent certificates are issued for their domains. The foundations of log-based PKI are public logs that wield append-only databases of X.509 certificates, to facilitate public auditing and provide efficient proofs on the presence of a specific entry in the log. These logs can then be monitored by a third party such as a domain owner to verify that no fraudulent certificates have been issued for their domains, and, in some cases, prevent browsers from accepting otherwise valid certificates that have not been logged.

The SSL Observatory [11] by the Electronic Frontier Foundation (EFF) is perhaps the first large-scale deployment of this concept, whereby a browser extension enables users to report previously unseen certificates to the EFF, which published anonymised versions of those logs. While the integrity of the EFF is widely acknowledged, this approach still remains based on a central repository.

Google's Certificate Transparency (CT) project [12,13] is an experimental protocol standardised by the Internet Engineering Task Force (IETF) that likewise aims to mitigate the threat of maliciously issued certificates by publicly logging certificates. Google's CT is a rather complex federated ecosystem, which introduces several new entities to the existing web PKI, starting with individual Submitters, who report certificates to vetted Loggers, who maintain a public append-only log periodically reviewed by Monitors, who can then report suspicious behavior, all under the watchful eye of Auditors, standalone or integrated into web clients.

Although trust in Google's CT is decentralised among the loggers, monitors and auditors, this protocol is unfortunately not as decentralised as one would like. For example, a misbehaving log server could create critical issues and lead to a single point-of-failure [14]. Another problem is the heavy reliance on third parties that monitor logs, combined with a near-total lack of incentives for third parties to actually do so. Further technical issues with Google's CT involve the lack of consistency if a particular domain decides to have multiple certificates (as Google itself does), and the absence of a mechanism for revoking certificates.

Other approaches akin to CT with log-based certificate management include ARPKI [15], AKI [16] and DTKI [17], with varying levels of centralisation.

1.2.2. Blockchain-Based PKI

A large proportion of recent projects seeking to rectify PKI trust issues are based on full redesigns, almost always involving a blockchain structure toward greater decentralisation. Unlike CT-based approaches, PKI redesigns will often be incompatible with existing infrastructure. Here, we mention several of these schemes. A comparison, including several other schemes, can be found in [18].

The notion of blockchain was first introduced as a public ledger of transactions in the Bitcoin cryptocurrency [19]. Unique combinations of properties make blockchains suitable for a variety of applications. For one, while the ledger itself is not distributed (it is replicated), its affirmation mechanism is based on a heavily decentralised consensus mechanism that makes it increasingly unfeasible to alter or delete previously time-stamped records. In principle, blockchains would provide ideal environments for decentralised PKIs—except that the replicated nature of a blockchain can make it impractical for large data sets. This is not expected to be a huge issue for DPKIT, as only the Merkle hash values are stored in the blockchain. (There are two distinct "extremely costly" aspects to cryptocurrencies such as Bitcoin: (1) there is the computational cost of the distributed consensus mechanism, often based on competitive proofs of work, needed to ensure permanence of the records; (2) there is also a storage cost, collectively borne by the users who dutifully replicate and keep a local copy of the entire ledger, to ensure the availability of those records).

Certcoin [20] is a decentralised PKI that builds PGP-like web-of-trust (WoT) identity retention on top of the Namecoin cryptocurrency for consistency enforcement. The protocol provides mechanisms for key registration, update, revocation, recovery, verification and lookup. However, it does not focus on fighting malicious users or verification and authentication of users. The lack of external identity validation might also be problematic when implemented in the real world.

Authcoin [21] is a proposal that focuses on the validation and authentication of public keys, rather than identity retention. It is based on a fault tolerant, replicated and transparent blockchain that aims to make it difficult for adversaries to introduce malicious certificates into the system. However, its reliance on interactive challenges and responses could be costly in performance, and raises credibility questions on the party to carry out validation and authentication.

Fredriksson's master's thesis [22] proposes a novel proof-of-stake protocol to build a decentralised PKI, with mappings between public keys and domains stored in a Merkle tree. Integrity is achieved by recording Merkle tree hashes in a blockchain, and having domain owners cross-sign their certificates. While this approach makes efficient use of a blockchain and other data structures, its requirement to have two extensions added to X.509 certificates, may be an insurmountable barrier to its practical adoption.

Another blockchain-based PKI scheme is CeCoin [23], which entirely removes the need for a trusted third party. Leveraging Merkle–Patricia [24] CeCoin aims to provide multi-certificates and identity assignment services. However, like its predecessors, CeCoin lacks a mechanism for external validation of identities, and its tabula rasa approach will likely preclude any real-world adoption.

Feng et al.'s proposal [25] implements a distributed PKI over a blockchain. New user registration is handled by a set of supervisory nodes, which are trusted authorities who validate registration requests. Cryptographic accumulators are used to efficiently construct identity witnesses for accessing services through third parties.

1.2.3. Traditional PKI Augmented with Blockchain

Several recent proposals, like ours, seek to improve the security of the existing PKI infrastructure using a blockchain as a decentralised means of publishing information. We introduce several of these here, and in Section 5.5, we give more details as we compare these schemes to our own.

CertLedger [26] provides additional functionality compared to our scheme, such as a mechanism for deciding ownership of a domain, and a centralised mechanism for determining which CAs are trusted. Madala et al. [27] propose Certificate Transparency using Blockchaing (CTB), which aims to augment certificate transparency by using a permissioned blockchain to store certificates. Wang et al. [28] propose blockchain-based certificate transparency, which is very similar to our scheme, but only domain owners can publish certificates. Zhao et al. [29] propose the notion of a CA proxy, implemented through smart contracts on a public blockchain, which serves to publicly broker transactions, like certificate signing requests. Yakubov et al. [30] propose to use a blockchain to store issued certificates and revocations, but without efficient or privacy preserving functionality for clients.

1.2.4. PKI Threat Models

The security of PKI is important enough that threat models have warranted standardisation of their own. The informational IETF draft "Attack Model for Certificate Transparency" [31] describes potential attack scenarios in a web context, categorising the threats into syntactic and semantic errors, and discusses their mitigation through transparency enhancement mechanisms.

We highlight that, in most cases, compromising a PKI is not the end goal of most attackers, who typically seek to obtain personal or trade-secret data, or gain control over a particular organisation for ulterior motives. Compromising the PKI is often a first step toward accessing the desired data. The main security property that any PKI must satisfy is preventing impersonation attacks, as it is one of the most compelling reasons for attacking a PKI.

1.3. Structure of the Paper

In the remainder of this article, we give the definition and prove the security of our scheme. In Section 2, we discuss some data structures that are necessary for our construction. Next, in Section 3, we discuss the various entities and their roles in the system, followed by a detailed construction for our system. Section 4 defines the security properties for our system and proves that they are satisfied. Section 5 discusses how our design goals have been achieved, considers the choice of ledger and how to motivate participants, and gives a comparison to previous schemes. Finally, Section 6 concludes the paper.

2. Preliminaries

Notation. We denote by E an ordered list of elements, where $[\ ]$ denotes the empty list. Indexing is 0-based: $E = [e_0, \ldots, e_{n-1}]$, and we write $E[i]$ to denote e_i and $E[i : j]$ to denote the sublist $[e_i, \ldots, e_{j-1}]$. $|E| = n$ is the length of E. We adopt the convention that $E[-1] = [\ ]$. We write $e \in E$ to indicate that an entry e is contained in the list E, i.e., $\exists i \in \mathbb{Z}\ e = E[i]$. For string concatenation, we use the notation $s \,||\, t$, which is also used for list concatenation, i.e., $[a, \ldots, z] \,||\, [a', \ldots, z'] = [a, \ldots, z, a', \ldots, z']$.

We use the notation $(orange : o, banana : b, apple : a)$ for dictionaries. If $d = (orange : o, banana : b, apple : a)$, we will use the notation $d.orange$ to refer to particular entries of the tuple. I.e., $d.orange$ will equal o is this example. We also assign values to keys in the dictionary using this notation, as in $d.orange \leftarrow 3$. The empty dictionary is denoted by $(\ )$.

Definition 1 (Collision resistance of Hash functions). *Let $\mathcal{M}$ be a set, let $\mathrm{H} : \mathcal{M} \to \{0,1\}^\lambda$ be an unkeyed hash function, and let $\mathbb{A}$ be a set of algorithms. We say that no $\mathcal{A} \in \mathbb{A}$ finds a pair (m, m') such that $m \neq m'$ and $\mathrm{H}(m) = \mathrm{H}(m')$ in time t with a probability higher than ϵ.*

For practical purposes, ϵ can be set to 2^{-128}.

2.1. Merkle Hash Trees, Binary Search Trees, and Merkle-BST

For our construction, we need an efficient method of indexing certificates and providing short proofs of membership. Hence we introduce Merkle binary search trees, which fill a similar role to Patricia tries used in many crypto-currencies, such as Ethereum [32]. In an abstract sense, they implement associative arrays with the additional property of providing proofs of membership.

2.1.1. Merkle Hash Trees

Ref. [33,34] were first described for authenticating large data sets. A Merkle tree is a useful cryptographic primitive to prove the existence of a record within a set E. Let $\mathrm{H} : \{0,1\}^* \to \{0,1\}^\lambda$ be a hash function. The tree is constructed by placing the values of E in the leaves of a binary tree and hashing each record. Each interior node is built by hashing the hashes of its two child nodes. The root of the tree, or root hash, acts as a fingerprint for the set E. One proves that a record exists in a (balanced) n-node tree by showing the $\mathcal{O}(log\ n)$ sibling hashes along its path, needed to reconstruct the root hash.

2.1.2. Binary Search Trees

(BST) , a.k.a., ordered binary trees, are a type of data structure that stores items sorted by its key and allows for fast lookup, insertion and removal of items [35]. To look for or insert a particular key, one traverses the tree from root to leaf, branching to left or right subtrees based on comparisons between the search key and the stored keys. This has expected time complexity $\mathcal{O}(log\ n)$ if the keys were inserted in random order, or the tree is explicitly balanced.

2.1.3. Merkle-BST and Security Properties

We introduce the Merkle-BST hybrid as a data structure which offers fast search, insertion, and update (fast removal is also possible, but not used in our application). along with short proofs of existence, for storing ordered sets of data. As in a BST, a Merkle-BST stores key-indexed entries in all nodes of a (randomly or explicitly balanced) binary tree, and overlays a Merkle-like hash structure for authentication; see Figure 2. The Merkle-BST data structure is useful as it provides a deterministic and remotely verifiable way of efficiently replicating large indexed sets of $(key, value)$ bindings, across many peers in a distributed system.

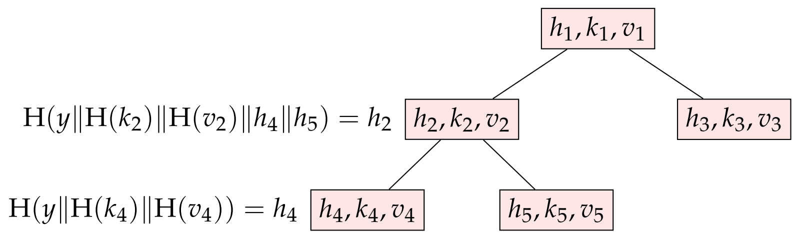
h_1, k_1, v_1
$\mathrm{H}(y\|\mathrm{H}(k_2)\|\mathrm{H}(v_2)\|h_4\|h_5) = h_2$
h_2, k_2, v_2
h_3, k_3, v_3
$\mathrm{H}(y\|\mathrm{H}(k_4)\|\mathrm{H}(v_4)) = h_4$
h_4, k_4, v_4
h_5, k_5, v_5

Figure 2. DPKIT Merkle Binary Search Tree. In each node, k_i is the search key (the domain name), v_i is a value associated with the key, and h_i is the hash associated with the node, computed as shown, where y is a "domain separation" label based on the node's position (see Section 2.1).

Nodes of a Merkle-BST have up to two children, and every node n stores a (k, v) key-value entry, along with a hash $\mathrm{H}(n)$. The hash is calculated by hashing the hashes of the children together with the node's entry, along with a special "domain-separation" flag $y \in \{1, 2, 3, 4\}$ describing the node's local topology (1 for leaves, 2 and 3 for nodes with a left or right child, and 4 for nodes with two children; unlike the Merkle hash tree, we allow our Merkle-BSTs to be unbalanced and have single-child nodes). For an interior node n, let $LC(n)$ and $RC(n)$ denote its left and right child, respectively. For a node with left and right child nodes, its hash is calculated as (see Figure 2 for the diagram and Figure A2 for the algorithm),

$$\mathrm{H}(n) = \mathrm{H}(4\|\mathrm{H}(k)\|\mathrm{H}(v)\|\mathrm{H}(LC(n))\|\mathrm{H}(RC(n)))$$

Complete algorithms are given in Appendix A.

Like binary search trees, Merkle-BSTs have efficient $(O(\log n))$ search, insert, and update operations when the keys are inserted in a reasonably random order. In our application, keys inserted as certificates are discovered, which will be reasonably random. This same condition means that the length of proofs of membership will be short $(O(\log n))$.

Lemma 1. *(Security of membership proofs in Merkle-BST (informal)) Let t be a Merkle-BST. Given that H is collision resistant, no adversary can efficiently find a proof that (k, v) exists in t if (k, v) has never been added to t, and k has never had its value updated to v in t.*

A rigorous statement is given in Appendix A. The proof, which we omit, is similar to the analogous security claim for Merkle trees.

2.2. Distributed Ledgers and Blockchains

A distributed ledger (e.g., blockchain), is a mechanism for de-centrally maintaining an append-only ledger, replicated among peers, and whose immutability of records is achieved by distributed consensus. The original and most famous example is the Bitcoin blockchain [19], secured without any central authority by miners competing for rewards using proofs of work; see, e.g., Ref. [36]. The appeal of work-secured blockchains is their ability to resist Sybil attacks (wherein a single user assumes multiple identities to overwhelm the control of legitimate users) without requiring an enrolment authority; see [37].

Our application does not require the full power of cryptocurrency blockchains, which also serve to (permanently) adjudicate mutually exclusive transactions. Rather, we view blockchains as black-box oracles that let users append small and infrequent amounts of data on a commonly shared (and replicated) timeline.

We formally define this notion of a distributed append-only ledger as the ideal functionality shown on Figure 3. An ideal functionality merely defines a set of oracles, or interface functions, with which participants can interact securely.

```
$\text{Init}_{\mathcal{L}}() \to l$:
1:    $L \leftarrow [\,]$
2:    return $\emptyset$

$\text{AddEntry}_{\mathcal{L}}(l, e)$:
1:    if $\text{Verify}_{\mathcal{L}}(e) = 0$ return $\perp$
2:    $L \leftarrow L \,||\, [e]$
3:    return $|L| - 1$

$\text{Handle}_{\mathcal{L}}(l)$:
1:    return $|L| - 1$

$\text{GetEntry}_{\mathcal{L}}(l, j)$:
1:    if $j \geq |L|$ return $\perp$
2:    return $L[j]$

$\text{GetAll}_{\mathcal{L}}(l)$:
1:    return $L$
```

Figure 3. Ideal functionality of an append-only ledger $\mathcal{L}$. The functionality is modelled as an oracle, with L the internal state of the oracle. Verify() is an application-specific verification function. The parameter l is a placeholder for information that would be required to access a real-world ledger protocol.

While append-only ledgers, in the form of blockchains, are frequently used in cryptocurrencies to prevent double spending, this is not in fact their core functionality, and it is not used in our constructions. Their primary function—establishing a canonical time ordered sequence of events—helps to prevent double spending by making it possible to determine canonically whether some specific funds have already been spent at the time when a new transaction is attempted. Append-only ledgers also facilitate our functionality by providing a canonical time ordered list of certificates recorded in the store, so that certificates cannot be removed, and root hashes so that proofs of membership can be easily verified.

There are many security considerations around ledgers that we do not wish to discuss in this paper, as they are more appropriate in a discussion of ledgers themselves. See, for example, Refs. [38–40]. For our purposes, we assume that the network of peers is large enough and sufficiently well connected that attacks that rely on controlling a large number of peers or their networks are not practical. More succinctly, we assume that peers all have access to the same ledger, and that their information is up to date so that we can consider the ledger as a single entity.

Depending on the desired security properties, an append-only ledger could be implemented by a trusted third party, a private blockchain, or a public blockchain. We defer this discussion to Section 5.2.

3. Decentralised PKI Transparency

In this section, we detail the construction, security, and use of our decentralised certificate store.

3.1. Entities, Operations and Functionalities

The basic entities of DPKIT are CAs, domain owners, servers, clients and peers. The impact on current PKI is less than some proposed work that we discussed in Section 1.2.2, as existing entities kept to handle basic functionalities of the system and only one new entity—peers—are added. Below, we summarise various interactions of the entities.

1. Certificate authorities: issue certificates for domains, and revocations.

2. Domain owners: obtain certificates from CAs, control servers, and (optionally) monitor the DPKIT for any suspicious activity.
3. Servers: allow clients to connect, and (optionally, if DPKIT-aware) request proofs of certificate membership from DPKIT peers.
4. Clients: initiate TLS connections with servers, accept or reject server certificates based on DPKIT information provided by the server and peers.
5. Peers: maintain the DPKIT data structure, record certificates, supply servers with proofs of certificate membership, provide auditing functionality.

Additionally, certificates can submitted to peers by any party, although in the honest case, this will likely be done by domain owners or certificate authorities.

The main functionality, which begins when a certificate is issued and ends with a TLS connection between a client and server, consists of these steps (see Figure 4):

1. A domain owner requests a certificate from a CA. The CA applies its vetting policies and issues a certificate to the domain owner (steps A and B).
2. Optionally, the domain owner or the CA sends the certificate to a DPKIT peer, who adds the certificate to the DPKIT data structure. The domain owner supplies the certificate to the domain TLS server (steps C and D).
3. A client requests a connection to the server. Optionally, the server requests a proof of certificate membership for its certificate from the DPKIT peer, which the peer provides. The server sends its certificate and optionally the proof to the client (steps E, F, G, H, I and J).
4. The client requests the latest root hash from a DPKIT peer and uses it to check the proof of certificate membership. It can also obtain the proof itself from the peer if not provided by the server. Depending on its security policy, the client may then complete the connection to the server (steps K, L and M).

Additionally, DPKIT provides the following functionalities:

- Revocation certificates may also be submitted to DPKIT peers. The revocation is recorded, and its presence is noted in proofs of certificate membership. Hence, clients will be made aware of any relevant revocations when verifying the proof of membership for a certificate.
- Any entity may submit a certificate to a DPKIT peer, and peers should add onto the DPKIT any missing valid certificates for which a proof is requested. This means that clients may contribute certificates that they have found which are missing from the DPKIT, increasing the probability that rare certificates are discovered and recorded.
- DPKIT peers also supply auditing functionality, including enumerations over all certificates for a domain, or over all domains, and indications whether certificates have been revoked.

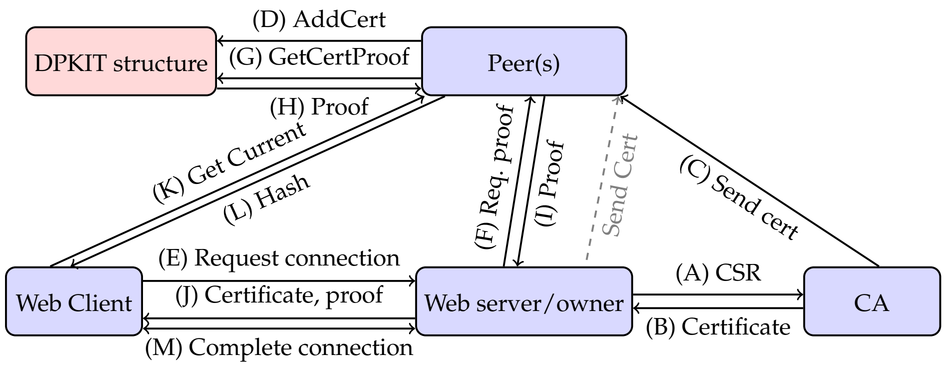

Figure 4. Overview over the interaction between entities involved in DPKIT scenario. See Section 3.1.

Definition 2. *(Certificate scheme) A certificate scheme consists of these algorithms, wherein c represents either a certificate or a revocation "certificate":*

- $\text{SubjectNames}_C(c) \rightarrow l$*: returns a list l of subject names for which the certificate c applies.*

- $\mathrm{FP}_C(c) \rightarrow \{0,1\}^\lambda$: *returns a fingerprint of the certificate or revocation.*
- $\mathrm{IsRevocation}_C(c) \rightarrow \{0,1\}$: *returns a bit indicating whether or not c is a revocation certificate.*
- $\mathrm{RevokedCert}_C(c) \rightarrow \{0,1\}^\lambda$: *if c is a revocation, returns the fingerprint of the certificate being revoked.*
- $\mathrm{IsValid}_C(c) \rightarrow \{0,1\}$: *returns a bit indicating whether or not this is a proper (e.g., X.509) certificate.*

Through the rest of the paper, we will assume that $\mathrm{FP}_C()$ is collision-resistant. For X.509 v.3 certificates, used for TLS on the web, SubjectNames() would return the domain name from "subject common name", as well as any additional domain names stated in the "subject alternative names" extension; $\mathrm{FP}_C()$ would return the hash of the certificate under a fixed cryptographic hash function.

Definition 3 (Decentralised PKI Transparency). *A decentralised PKI transparency scheme (DPKIT) θ for a certificate scheme consists of these algorithms, where d is a data structure representing the overall state of the system:*

- $\mathrm{Init}_D() \rightarrow d$*: deterministic algorithm that outputs an initial data structure d.*
- $\mathrm{AddCert}_D(d, c) \rightarrow d'$ *or* $\perp$*: deterministic algorithm taking a certificate c and a data structure d as input and outputs an updated data structure d′ or ⊥ if not valid.*
- $\mathrm{Search}_D(d, N) \rightarrow s$ *or* $\emptyset$*: deterministic algorithm that takes a data structure d and a subject name N and outputs a data structure representing all certificates for N or ∅ if N is not present.*
- $\mathrm{SearchCert}_D(s, F) \rightarrow (c, r)$ *or* $\perp$*: deterministic algorithm that takes a data structure s and a certificate fingerprint F and outputs a certificate and revocation tuple (c, r), either of which may be ⊥, or ⊥ if both not present.*
- $\mathrm{GetCertProof}_D(d, c) \rightarrow P$ *or* $\perp$*: deterministic algorithm that takes a certificate c and a data structure d and outputs a membership proof P or an error ⊥.*
- $\mathrm{GetAllCerts}_D(d) \rightarrow E$ *or* $\perp$*: deterministic algorithm that takes an object d and outputs an ordered list of entries or an error symbol ⊥.*
- $\mathrm{GetFingerprint}_D(d) \rightarrow F$*: deterministic algorithm that takes a data structure d and returns a fingerprint F that captures all entries in the structure.*
- $\mathrm{VerifyCertProof}_D(P, F, c) \rightarrow \{0,1\}$*: deterministic algorithm that, given a certificate c, fingerprint F and membership proof P, outputs a bit $b \in \{0,1\}$.*
- $\mathrm{TestRevoke}_D(P) \rightarrow \{0,1\}$*: deterministic algorithm that takes as input a membership proof P and outputs a bit $b \in \{0,1\}$.*

The algorithms defined in the above Definitions 2 and 3 can be used to carry out all the interactions between peers/clients and the DPKIT structure, as described in Figure 4, which includes inserting and auditing certificates in the system.

3.2. DPKIT Construction

Now that we have the required associative array and append-only ledger, we can construct our associative distributed ledger for PKI transparency, DPKIT, as specified in Figure 5.

The DPKIT of Figure 5 stores domain names as keys and certificates as values in a Merkle-BST, accessed through the functions defined in Figure A2.

In our application, the main search key k is a domain name, and the value v is a pointer to its very own data structure; a secondary Merkle-BST (as illustrated in Figure 1). This certificate tree is unique to every node, and stores all the certificates that pertain to that particular domain name k.

All these certificate updates contribute to the root hash of the certificate Merkle-BST, and from there, the hash of the domain-name node in the primary Merkle-BST, and in turn, the primary root hash.

```
$\text{Init}_D() \to d$:
1:  $d \leftarrow (\,)$
2:  $d.ledger \leftarrow \text{Init}_{\mathcal{L}}()$
3:  $d.tree \leftarrow \text{Init}_M(\text{FP}_S, \text{H})$
4:  return $d$

$\text{Search}_D(d, N) \to s$:
1:  $s \leftarrow \text{Search}_M(d.tree, N)$
2:  return $s$

$\text{SearchCert}_D(s, F) \to (c, r)$:
1:  $q \leftarrow \text{Search}_M(s, F)$
2:  if $q = \perp$ return $\perp$
3:  $(c, r) \leftarrow q$
4:  return $(c, r)$

$\text{GetCertProof}_D(d, c) \to P$:
1:  $N \leftarrow \text{SubjectNames}_C(c)[0]$
2:  $s \leftarrow \text{Search}_D(d.tree, N)$
3:  if $s = \emptyset$ return $\perp$
4:  $q \leftarrow \text{SearchCert}_D(s, \text{FP}_C(c))$
5:  if $q = \perp$ return $\perp$
6:  $(c', r) \leftarrow q$
7:  if $c \neq c'$ return $\perp$
8:  $p_1 \leftarrow \text{GetProof}_M(d.tree, N)$
9:  $p_2 \leftarrow \text{GetProof}_M(s, \text{FP}_C(c))$
10: if $r \neq \emptyset, f_r \leftarrow \text{FP}_C(r)$
11: else $f_r \leftarrow \emptyset$
12: $f_c \leftarrow \text{FP}_C(c)$
13: $f_s \leftarrow \text{GetFingerprint}_M(s)$
14: return $(p_1, p_2, f_r, f_c, f_s)$

$\text{GetAllCerts}_D(d) \to E$:
1:  $E \leftarrow [\,]$
2:  $L \leftarrow \text{GetAll}_{\mathcal{L}}(d.ledger)$
3:  for $n = 0$ to $Length(L) - 1$
4:      $(F, c) \leftarrow L[n]$
5:      $E.append(c)$
6:  return $E$

$\text{GetFingerprint}_D(d) \to F$:
1:  $(F, c) \leftarrow \text{GetEntry}_{\mathcal{L}}(d.ledger, \text{Handle}_{\mathcal{L}}())$
2:  return $F$

$\text{FP}_D(v) \to h$:
1:  $(c, r) \leftarrow v$
2:  if $(c = \emptyset)$ $ch = \emptyset$
3:  else $ch = \text{FP}_C(c)$
4:  if $(r = \emptyset)$ $rh = \emptyset$
5:  else $rh = \text{FP}_C(r)$
6:  return $\text{FPN}_D(ch, rh)$

$\text{FPN}_D(ch, rh) \to h$:
1:  if $(ch \neq \emptyset \wedge rh \neq \emptyset)$ return $H(1 \mid\mid ch \mid\mid rh)$
2:  if $(ch \neq \emptyset)$ return $H(2 \mid\mid ch)$
3:  if $(rh \neq \emptyset)$ return $H(3 \mid\mid rh)$
4:  return $\perp$

$\text{FP}_S(s) \to h$:
1:  return $\text{GetFingerprint}_M(s)$
```

```
$\text{AddCert}_D(d, c) \to d'$:
1:  if $\text{IsValid}_C(c) = 0$ return $\perp$
2:  $d' \leftarrow d$
3:  if $\text{IsRevocation}_C(c) = 0$
4:      $k \leftarrow \text{FP}_C(c), v \leftarrow (c, \emptyset)$
5:  else
6:      $k \leftarrow \text{RevokedCert}_C(c), v \leftarrow (\emptyset, c)$
7:  for $N$ in $\text{SubjectNames}_C(c)$
8:      $s \leftarrow \text{Search}_D(d, N)$
9:      if $s = \emptyset$
10:         $s \leftarrow \text{Init}_M(\text{FP}_D, \text{H})$
11:         $\text{Insert}_M(s, k, v)$
12:         $\text{Insert}_M(d'.tree, N, s)$
13:     else
14:         $q \leftarrow \text{SearchCert}_D(s, k)$
15:         if $q = \perp$
16:             $\text{Insert}_M(s, k, v)$
17:         else
18:             $(c', r') \leftarrow q$
19:             if $\text{IsRevocation}_C(c) = 0$
20:                 if $c' \neq \emptyset$ return $\perp$
21:                 $\text{Update}_M(s, k, (c, r'))$
22:             else
23:                 if $r' \neq \emptyset$ return $\perp$
24:                 $\text{Update}_M(s, k, (c', c))$
25:         $d'.tree \leftarrow \text{Update}_M(d'.tree, N, s)$
26: $L \leftarrow \text{AddEntry}_{\mathcal{L}}(d.ledger, (\text{GetFingerprint}_M(d'), c))$
27: return $d'$

$\text{VerifyCertProof}_D(P, F, c) \to b$:
1:  $(p_1, p_2, f_r, f_c, f_s) \leftarrow P$
2:  $n \leftarrow \text{SubjectNames}_C(c)[0]$
3:  if $\text{VerifyProof}_M(p_1, F, H(n), f_s, H) = 0$
4:      return 0
5:  $f_{cr} \leftarrow \text{FPN}_D(f_c, f_r)$
6:  if $\text{VerifyProof}_M(p_2, f_s, H(\text{FP}_C(c)), f_{cr}, H) = 0$
7:      return 0
8:  return 1

$\text{TestRevoke}_D(P) \to b$:
1:  if $P.f_r \neq \emptyset$ return 1
2:  return 0

$\text{Verify}_L(d, (F, c))$:
1:  Run lines 1-25 of $\text{AddCert}_D(d, c)$
2:  if $\text{GetFingerprint}_M(d'.tree) = F$
3:      return 1
4:  return 0
```

Figure 5. Our construction of a DPKIT scheme (D) from a associative array with membership proofs (M) and a ledger ($\mathcal{L}$). $\text{Verify}_L()$ is used by $\mathcal{L}$ to verify entries as part of AddEntry(). H is a hash function. Note that $\mathcal{L}$ (i.e., the peers maintaining the ledger) must maintain a copy of d (which we do not model here) or reconstruct d from the previous entries in the ledger. The main data structure is $d = (ledger : l, tree : t)$, where l is a handle for the ledger, and t is a MerkleBST data structure.

Some certificates allow for a primary domain name and additional subject alternative names in a single certificate. For example, there could be www.gmail.com, www.mail.google.com etc. for *Gmail*. These various domain names will be represented by separate nodes in the main tree. If a certificate has multiple subject names, then it will be stored in each of their certificate trees. (As an implementation detail, the certificates can be stored in a separate location, with only pointers stored in the certificate tree. This allows for a certificate to be stored only once when it has multiple subject names).

Whenever a new certificate is inserted into the trees, the root hash of the primary tree, along with the added certificate, is added to the ledger.

4. Security Analysis and Evaluation

In this section, we show security results on decentralised PKI transparency, namely that it satisfies the three properties of entry non-removability, revocation reveal and proof consistency from Figure 6.

4.1. Security Goals

In DPKIT, the overall security goal is to ensure that information about certificates (including revocations) is not altered or removed. Thus, we focus on threats where the attacker presents false information to the "verifier"—which can be a client or a peer in our terminology—in an attempt to disrupt normal operations.

The three security properties formally specified in Figure 6 capture all the ways in which a malicious entity may be able to pass conflicting information. We say that the scheme has a security property of the corresponding security game and cannot be won with probability more that some security parameter ϵ by any polynomial time $\mathcal{A}$. We name and describe the three properties as follows:

Non-removability, per Experiment non-removable, demands that it is hard for an adversary to remove an entry once logged through honest functionalities.

Proof consistency, per Experiment entry-proof, entails that it is hard for an adversary to provide a valid membership proof for an entry that is invalid or not yet logged.

Revocation reveal, per Experiment show-revoke, demands that it is hard for an adversary to hide an entry's revocation information if it exists in the log.

A scheme that satisfies the above ensures that a malicious verifier cannot make peers interpret different things about entries covered by a root hash or a proof.

We consider security in an "ideal-ledger model", wherein the append-only ledger is modelled using the oracles of the ideal ledger functionality given in Figure 3. We follow a provable security game-based approach to define and prove DPKIT security properties. The three security properties are each formalised as a game against a malicious entity, which "plays" in a security experiment that acts as the verifier that the attacker is trying to deceive.

```
$\mathrm{Exp}^{\text{entry-proof}}_{\theta,\mathcal{L}}(\mathcal{A})$:
1:  $\mathrm{Init}_{\mathcal{L}}()$
2:  $d \leftarrow \mathrm{Init}^{\mathcal{L}}_{\theta}()$
3:  $(P, c) \leftarrow \mathcal{A}()^{\theta*(d,\cdot),\mathcal{L}}$
4:  $F \leftarrow \mathrm{GetFingerprint}^{\mathcal{L}}_{\theta}(d)$
5:  if $\mathrm{VerifyCertProof}_{\theta}(P, F, c) = 0$ return 0
6:  for $N$ in $\mathrm{SubjectNames}(c)$;
7:      $s \leftarrow \mathrm{Search}^{\mathcal{L}}_{\theta}(d, N)$
8:      if $s = \emptyset$, return 1
9:      $(c', r') \leftarrow \mathrm{SearchCert}^{\mathcal{L}}_{\theta}(s, \mathrm{FP}(c))$
10:     if $c' \neq c$, return 1
11: return 0
```

```
$\mathrm{Exp}^{\text{show-revoke}}_{\theta,\mathcal{L}}(\mathcal{A})$:
1:  $\mathrm{Init}_{\mathcal{L}}()$
2:  $d \leftarrow \mathrm{Init}^{\mathcal{L}}_{\theta}()$
3:  $(c, r) \leftarrow \mathcal{A}_1()^{\theta*(d,\cdot),\mathcal{L}}$
4:  if $\mathrm{RevokedCert}_C(r) \neq \mathrm{FP}(c)$, return 0
5:  $d \leftarrow \mathrm{AddCert}^{\mathcal{L}}_{\theta}(d, r)$
6:  $P \leftarrow \mathcal{A}_2()^{\theta*(d,\cdot),\mathcal{L}}$
7:  $F \leftarrow \mathrm{GetFingerprint}^{\mathcal{L}}_{\theta}(d)$
8:  if $\mathrm{VerifyCertProof}^{\mathcal{L}}_{\theta}(P, F, c) = 0$ return 0
9:  if $\mathrm{TestRevoke}^{\mathcal{L}}_{\theta}(P) = 1$ return 0
10: return 1
```

```
$\mathrm{Exp}^{\text{non-removable}}_{\theta,\mathcal{L}}(\mathcal{A})$:
1:  $\mathrm{Init}_{\mathcal{L}}()$
2:  $d \leftarrow \mathrm{Init}^{\mathcal{L}}_{\theta}()$
3:  $(c, st) \leftarrow \mathcal{A}_1()^{\theta*(d,\cdot),\mathcal{L}}$
4:  $d \leftarrow \mathrm{AddCert}^{\mathcal{L}}_{\theta}(d, c)$
5:  if $(d = \perp)$ return 0
6:  $\mathcal{A}_2(st)^{\theta*(d,\cdot),\mathcal{L}}$
7:  if $c \notin \mathrm{GetAllCerts}^{\mathcal{L}}_{\theta}(d)$, return 1
8:  for $N$ in $\mathrm{SubjectNames}(c)$;
9:      $s \leftarrow \mathrm{Search}^{\mathcal{L}}_{\theta}(d, N)$
10:     if $s = \emptyset$, return 1
11:     if $\mathrm{IsRevocation}(c) = 0$
12:         $q \leftarrow \mathrm{SearchCert}^{\mathcal{L}}_{\theta}(s, \mathrm{FP}(c))$
13:         if $(q = \perp)$ return 1
14:         $(c', r') \leftarrow q$
15:         if $c' \neq c$, return 1
16:     else
17:         $(c', r') \leftarrow \mathrm{SearchCert}^{\mathcal{L}}_{\theta}(s, \mathrm{RevokedCert}(c))$
18:         if $(q = \perp)$ return 1
19:         $(c', r') \leftarrow q$
20:         if $r' \neq c$, return 1
21: return 0
```

Figure 6. Security properties of a DPKIT scheme θ, using append only ledger $\mathcal{L}$, against a malicious $\mathcal{A}$. θ represents the (possibly restricted) tuple of DPKIT algorithms that the adversary is given access to. $\mathcal{L}$ stands for an instantiation of the ideal append-only ledger functionality defined in Figure 3. We write $\theta * (d, .)$ and $\theta * (d, .).\mathrm{Alg}$ to indicate that $\mathcal{A}$ has oracle access to all or a specific function(s) in θ, invoked with the data structure d as a fixed parameter. For example, $\theta * (d, .) = (\mathrm{AddCert}, \mathrm{Search}, ..)$.

The parameter d indicates that these are the functions that the adversary can use to manipulate the global system state d, where it is implied that these manipulations may have side effects, as the global state can be updated. Note that, since d is anchored to an underlying append-only ledger, and since in our model it is not possible to remove entries from an append-only ledger, the adversary is not given access to such hypothetical functions.

With these experiments in place, we can now define the security and correctness of a DPKIT scheme.

Definition 4. *Let $\mathbb{A}$ be a set of algorithms. We say that a DPKIT scheme θ is ϵ-secure with respect to $\mathbb{A}$ under the ideal-ledger assumption if, for all $\mathcal{A} \in \mathbb{A}$,*

$$\Pr[\mathrm{Exp}^{\text{non-removable}}_{\theta,\mathcal{L}}(\mathcal{A}) = 1] \leq \epsilon \tag{1}$$
$$\Pr[\mathrm{Exp}^{\text{entry-proof}}_{\theta,\mathcal{L}}(\mathcal{A}) = 1] \leq \epsilon \tag{2}$$
$$\Pr[\mathrm{Exp}^{\text{show-revoke}}_{\theta,\mathcal{L}}(\mathcal{A}) = 1] \leq \epsilon \tag{3}$$

where $\mathcal{L}$ is an ideal-ledger oracle functionality, as defined in Figure 3.

Definition 5. *A DPKIT scheme θ is correct, provided that:*

1. *If $\mathrm{AddCert}_{\theta,\mathcal{L}}(d, c)$ has not been called on certificate c, then $\forall n \in \mathrm{SubjectNames}(c)$*

$$\mathrm{SearchCert}_{\theta,\mathcal{L}}(\mathrm{Search}_{\theta,\mathcal{L}}(d, n), FP(c)) \neq (c, \cdot) \tag{4}$$
$$c \notin \mathrm{GetAllCerts}_{\theta,\mathcal{L}}(d) \tag{5}$$

2. *If $\mathrm{AddCert}_{\theta,\mathcal{L}}(d, c)$ has been called on a certificate c, then we always have*

$$\mathrm{VerifyCertProof}_{\theta,\mathcal{L}}(\mathrm{GetCertProof}_{\theta,\mathcal{L}}(d, c), \mathrm{GetFingerprint}_{\theta,\mathcal{L}}(d), c) = 1.$$

3. *For a certificate c, if* $\mathrm{AddCert}_{\theta,\mathcal{L}}(d,r)$ *has never been called on any revocation r bearing onto c, that is, s.t.* $\mathrm{RevokedCert}(r) = FP(c)$*, then we always have*

$$\mathrm{TestRevoke}_{\theta,\mathcal{L}}(\mathrm{GetCertProof}_{\theta,\mathcal{L}}(d,c)) = 0.$$

4.2. Proofs of Security

We first note that the scheme is correct.

Lemma 2 (Correctness of DPKIT scheme). *The DPKIT scheme θ described in Figure 5 is correct.*

Lemma 2 follows straightforwardly from the definition of the protocol and the properties of MBSTs.

Theorem 1. *(Non-removability) If hash function* H *and fingerprint function* FP_C *are collision-resistant, then in DPKIT scheme θ (with hash function* H *and fingerprint function* FP_C*) in ideal ledger model $\mathcal{L}$, no malicious entity can present two different entries for the same fingerprint. More precisely, no $\mathcal{A}$ wins* $\mathrm{Exp}^{\text{non-removable}}_{\theta,\mathcal{L}}(\mathcal{A})$*, with probability higher than ϵ. That is,*

$$Pr[\mathrm{Exp}^{\text{non-removable}}_{\theta,\mathcal{L}}(\mathcal{A}) = 1] \leq \epsilon.$$

Proof. The adversary can only manipulate the data structure through θ and $\mathcal{L}$. The challenger simulates all functions in the oracle for the adversary. We show that a successful adversary $\mathcal{A}$ effectively removes an entry after it was added to the Merkle BST, which leads to a contradiction with append only ledger $\mathcal{L}$ or with Lemma 1, or produces a collision in H or FP_C.

First note that the call in line 4 of $\mathrm{Exp}^{\text{non-removable}}$ succeeds, otherwise $\mathcal{A}$ loses in line 5. Now, suppose $\mathcal{A}$ wins $\mathrm{Exp}^{\text{non-removable}}_{\theta,\mathcal{L}}(\mathcal{A})$ from the case in line 7. In line 4, c is added without error to d and $\mathcal{L}$, meaning that $(d.tree.hash, c)$ exists as an entry somewhere in $\mathcal{L}$ and will again be in the list $\mathrm{GetAllCerts}^{\mathcal{L}}_{\theta}(d)$ as per its algorithm. This produces a contradiction.

Suppose $\mathcal{A}$ wins $\mathrm{Exp}^{\text{non-removable}}_{\theta,\mathcal{L}}(\mathcal{A})$ from the case in line 10. By correctness property of DPKIT in Definition 5, when AddCert is called, the entry will be added and subsequent Search will return a value of the entry. However, line 9 implies an empty value. This is a contradiction on oracle functionalities, meaning that no $\mathcal{A}$ will win here either. Similarly, $\mathcal{A}$ cannot win in lines 13 or 18.

Finally, suppose $\mathcal{A}$ wins $\mathrm{Exp}^{\text{non-removable}}_{\theta,\mathcal{L}}(\mathcal{A})$ from the case in lines 15 or 20. In the first case, c' is in the tree for key $\mathrm{FP}_C(c)$, which can only happen if $\mathrm{FP}_C(c) = \mathrm{FP}_C(c')$, implying a collision in FP_C. In the second case, c is a revocation which would have been added to s under the fingerprint of the certificate that it revokes, but it is missing. This cannot happen, since it was added in line 4. □

Theorem 2 (Proof consistency). *If hash function* H *and fingerprint function* FP_C *are collision-resistant, then in DPKIT scheme θ (with hash function* H *and fingerprint function* FP_C*) in the ideal ledger model $\mathcal{L}$, no malicious entity can present a proof for an entry which is invalid or not logged. More precisely, no $\mathcal{A}$ wins* $\mathrm{Exp}^{\text{entry-proof}}_{\theta,\mathcal{L}}(\mathcal{A})$*, with probability higher than ϵ. That is,*

$$Pr[\mathrm{Exp}^{\text{entry-proof}}_{\theta,\mathcal{L}}(\mathcal{A}) = 1] \leq \epsilon.$$

Proof. We show that if $\mathcal{A}$ produces a valid proof for an entry which does not exist in the log, then this produces a contradiction with Lemma 1. We assume that no hash or fingerprint collisions are present. This is true except with probability at most ϵ.

Let $P = (p_1, p_2, f_r, f_c, f_s)$ as returned in line 3 of $\mathrm{Exp}^{\text{entry-proof}}$. F is the root hash of a Merkle-BST tree which, since F is on $\mathcal{L}$ and has been verified by Verify, was constructed honestly. Note that $\mathrm{VerifyCertProof}_D$ has returned 1, so p_1 was verified in line 3 and by

Lemma 1 n is a key in the primary tree of d with value f_s the hash of some secondary tree. Similarly, since p_2 is accepted as a valid proof in line 6 we see that secondary tree with root hash f_s contains a key $\mathrm{FP}_C(c)$ with value (c, r) for some r.

Now for $\mathcal{A}$ to win in line 8, c must have some subject name N that does not exist as a key in the primary tree. However, from the above discussion, we know that c was indeed added for at least one subject name, and since the trees were constructed honestly, c would have been added to all of its subject name subtrees, so those subject names must exist as keys. Hence, $\mathcal{A}$ cannot win here.

Similarly, for $\mathcal{A}$ to win in line 10, c must not be entered in some secondary tree for one of its subject names, but this cannot happen, because the trees were constructed honestly.

In all cases, it is impossible for $\mathcal{A}$ to win, except by producing a hash or fingerprint collision or by forging an Merkle-BST proof, both of which occurs with probability at most ϵ. □

Theorem 3. *(Revocation reveal) If hash function* H *is collision-resistant, then in DPKIT scheme θ (with hash function H) in ideal ledger model $\mathcal{L}$, when a client requests a proof for an entry, revocation information should be included if there exists any corresponding revocation certificate. In other words, no malicious entity should be able to hide the revocation certificate of a particular entry. More precisely, no $\mathcal{A}$ wins $\mathrm{Exp}^{\mathsf{show\text{-}revoke}}_{\theta,\mathcal{L}}(\mathcal{A})$, with probability higher than ϵ.*

$$Pr[\mathrm{Exp}^{\mathsf{show\text{-}revoke}}_{\theta,\mathcal{L}}(\mathcal{A}) = 1] \leq \epsilon.$$

Proof. First, since any invalid proof is rejected in line 8 of $\mathrm{Exp}^{\mathsf{show\text{-}revoke}}$, we may assume that c is indeed logged honestly. We also know that r was logged honestly in line 5. However, by Lemma 1, in order for the proof to be accepted as valid, the hash f_{cr} must be correct in line 6 of $\mathrm{VerifyCertProof}_D$. f_{cr} is produced by FPN_D and will only be the correct value if the correct value of f_r appears in the proof. Hence, $f_r \neq \emptyset$ and $\mathrm{TestRevoke}_D$ thus returns 1. Hence, $\mathcal{A}$ cannot win, except with probability ϵ by producing a collision. □

5. Discussion

5.1. Design Goals Revisited

Here, we revisit our design goals from Section 1.1.1 and show how each of them is satisfied by our construction.

- *Transparency*: When implemented using a public blockchain for the ledger, all information is publicly available. When proper incentives are used, certificates and revocations will be aggressively added to the ledger when they begin to circulate, meaning that maximum information is available in one common, publicly accessible place. Certificates and revocations cannot be removed from the ledger (non-removable entry security property). Additionally, participating clients obtain guarantees that certificates have been logged (proof consistency property).
- *Multiple certificates*: This is handled by using certificate subtrees for each subject name
- *Multiple domain names*: This is handled by adding each certificate to the tree under all of its subject names
- *No single point of failure*: This is achieved by through the decentralized ledger, and hence is sensitive to the ledger used.
- *Scalability*: The efficiency of operations and proofs in Merkle-BSTs allows for efficient operations. The scalability of the system as a whole thus depends mostly on the ledger being used. Note that issuing certificates happens at a much slower pace than financial transactions, so the scalability needs are much more modest than what is required for crypto-currency ledgers.
- *Efficiency*: TLS clients and servers need only to process proofs of membership, which are short. Hence, the impact is low. Servers can cache proofs for greater efficiency.
- *Client privacy*: The only information revealed by a client to other entities is to peers, from which they only obtain recent root hashes, and hence only reveal their participa-

tion in the scheme, and TLS servers, which they need to contact anyways to access their services.

- *Revocation*: Revocations are stored, and proofs of membership reveal their existence to clients (Revocation reveal security property.)

5.2. Considerations around Ledgers and Participants

5.2.1. Participants and Incentives

Let us consider the incentives of the various participants:

- *Clients:* Clients have two main goals in our model: to avoid impersonation attacks and to avoid leaking information to third parties. While the first is best served by having a complete record of certificates and revocations, a client may choose not to log new certificates that they see to avoid leaking their browsing information to peers.
- *Domain owners and servers:* Domain owners have an incentive to prevent impersonators from capturing business or driving away customers. Again, this is served by a complete record, but domain owners are only interested in their own domains, and hence only have an incentive to report certificates for their domains. In some circumstances, they may be incentivized to suppress knowledge of successful attacks—possibly to prevent harm to their reputation—and so may in fact not want false certificates to be logged. If clients insist on only connecting to sites with logged certificates (much as today they more or less insist on only connecting to sites with valid certificates), then domain owners will be strongly incentivized to have their legitimate certificates logged or lose business.
- *Certificate authorities:* Certificate authorities, as trusted entities, only stay in business for as long as they remain trustworthy. While participation in logging schemes may improve the perception of their security, any evidence of security failures reduces their perceived trustworthiness and hence disincentivizes logging problematic certificates. If domain owners insist on having their legitimate certificates logged, then certificate authorities will be incentivized to do this or lose business.
- *Peers:* Peers' main role is to contribute to the storage and computational power required to maintain the ledger. Although they may also play one of the above roles, the role of peer does not bring with it any special incentives.

5.2.2. Maintaining the Ledger

In order for our scheme to be useful, we need two main things to happen: the ledger must be maintained, and certificates must be contributed. Let us concentrate on the latter first. The path to having legitimate certificates logged is relatively straightforward: clients can insist that certificates are logged or they will not connect. However, getting illegitimate certificates logged is more problematic, as clients are the only ones with a strong incentive to do this, and it is offset by desires for privacy. There are some ways around this, such as anonymising networks, but this increases the complexity. A better way might be to have bounties for newly submitted certificates. If using an existing cryptocurrency blockchain with smart contracts (e.g., Ethereum) to implement the ledger, then smart contracts can be issued that pay a user when they submit a previously unsubmitted certificate or revocation to the log. With this type of incentive in place, revocations and fraudulent certificates are much more likely to be quickly submitted, making it possible to issue revocations or take action as required. To avoid pathological activity, such as creating certificates in bulk purely to claim the bounty, in will be necessary to place restrictions on these smart contracts, for example making them pay out only for certificates signed by certain certificate authorities, or issued for certain domains.

The remaining question is how to pay for bounties and also how to incentivize peers. If using an existing blockchain, then the latter is already taken care of, although it will likely be necessary to pay transaction fees for any changes to the ledger, i.e., when adding certificates or revocations. Transaction fees are similar to bounties in that they are both paid for adding certificates. There are a few possibilities for who might be willing to pay for certificates to be added:

- Groups of clients, web browser vendors, governments, or other organisations that have an overall desire for security on (parts of) the Internet
- Domain owners or Certificate authorities who are forced to log their own legitimate certificates by client policies
- Domain owners who wish to know about fraudulent certificates for their domains so that they can take necessary security actions
- Certificate authorities who are interested in testing the security claims of their competitors
- Certificate authorities who wish to present a positive security image by advertising proactive security policies
- Insurance agencies who insure companies against cyber threats, that want to mitigate damage as quickly as possible, and who want to catch untrustworthy certificate authorities who increase the chances of security threats against their insurees.

Since bounties need to be public, another possibility arises: if clients insist, not only that legitimate certificates are logged, but also that bounties exist for all certificates for the related subject names, then they can force domain owners to create such bounties. Domain owners may then force certificate authorities to do this on their behalf. In this way, the cost of maintaining the ledger can be placed with the certificate authorities, likely funded through fees paid for obtaining certificates. Such a possibility requires a large mass of clients working together.

If not using an existing ledger, then the ledger must be maintained and this will incur a cost, especially if proof-of-work is used. This problem is not unique to DPKIT, however, as all other CT systems still rely on a ledger or centralised servers which must be maintained. The default stance seems to be that these costs will be covered by various major players, such as CAs, browser vendors or other large organisations with an interest in keeping the internet secure. Having a decentralised system such as ours actually makes this type of funding easier, since anyone with an interest can contribute by becoming a peer, without needing to collaborate with, or even trust, other parties in the system.

One last service that must be provided is that of providing membership proofs to servers and clients. These operations do not require ledger transactions, and hence may be relatively inexpensive, but the infrastructure must still exist. Peers are the natural entities to provide these services as they already maintain the necessary data. Likely these services are too frequent and trivial resource-wise to pay for them through mechanisms like micropayments, but a subscription model would be appropriate, with the domain owners paying for their servers' to access peers. Another possibility would be for this service to be provided by CAs running their own peers and paid for through certificate fees.

5.3. Usage

5.3.1. What Goes on the Ledger?

In order to be useful for security purposes, the certificate store must facilitate the detection of fraudulent certificates. There are a few scenarios:

- A certificate appears in the store that the domain owner did not apply for. This check must be done by the domain owner or someone with sufficient knowledge to recognise certificates that they have requested. In order for the certificate store to be useful in this manner, as many certificates as possible must be logged, since those collecting certificates do not typically have the required knowledge to decide fraudulent certificates from authentic ones.
- A certificate is presented to a client that does not exist on the log. In this case, the client may decide not to connect. A certificate that is not logged may mean that the domain owner is not aware of it and hence not able to take appropriate actions against the certificate, such as seeking a revocation from the issuing CA. Hence, it should be standard practice to log certificates as soon as they are issued.

- A certificate is presented to a client whose revocation exists on the log. In this case, the client should not connect. In order for this to be useful, as many revocations as possible must be logged, whether or not the corresponding certificate has been. Hence, it should be standard practice to log revocations soon as they are issued.

Additionally, there is the question of whether only certificates from certain CAs should be logged, or if all certificates should be logged, regardless of who signed them. While the latter invites the possibility of denial of service by attackers who create their own certificates to flood the ledger, the former requires some decision about which CAs to log, which would be difficult without some kind of centralisation. However, denial of service is unlikely when an existing ledger such as Ethereum is used where adding certificates incurs a fee. Assuming that bounties are used to offset this fee, the selectivity of those bounties, and the fact that CAs control their own certificate issuance, means that flooding will be unlikely (unless a CA decides to clean up in the short term on bounties paid for by another party! For this reason, bounties should have limits).

To eliminate the problem almost entirely, at the cost of higher complexity, separate DPKIT instances could be created for each CA, accepting only valid certificates from that CA. This may not be difficult to implement if an existing blockchain is used, so that creating a new instance is relatively inexpensive and there is no cost associated with maintaining each instance, aside from fees to add certificates. Clients and other parties are then free to ignore instances for any CA that they are not interested in.

Because of these considerations, we recommend that all certificates are logged, as long as they are valid, regardless of the signing CA.

5.3.2. Privacy

The privacy property of our scheme is that clients do not need to reveal what domains they connect to in order to benefit. However, it is still true that security is highest when as many certificates as possible are logged. Hence, it would be beneficial for clients to contribute certificates, which are not anonymous in our system. Some considerations for this problem are:

- In the honest case, which is the vast majority of traffic, certificates will already be logged and there is no need for clients to submit certificates. Hence, for most clients, it is rare to encounter an unlogged certificate.
- A client may opt to contribute an unlogged certificate when they find one. The client may prompt the user to make a decision, and it may be the case that most people will choose to log the certificate in most scenarios.
- An anonymising layer, similar to TOR, may be added to the communication between clients and peers. Such a system could be simple to implement, since there is no need for the client to receive a reply from the peer. It may be that blockchain technologies will incorporate such anonymous transactions themselves in the near future, with some already providing some level of anonymity [40].
- If bounties are used, as proposed above, clients may willingly accept the fee for their breach of privacy, or some clients may go out of their way to collect and report certificates in order to collect bounties. However, bounty hunters will likely not encounter certificates that are part of attacks targeted at particular users.
- If a fraudulent certificate is used in a targeted attack on a small set of clients, then the extent of the attack is necessarily limited. While not ideal, an unlogged certificate in this scenario will do little immediate damage. However, the more clients are involved, or the more fraudulent certificates that are issued in the same manner, the more likely it is that at least one client is willing to log the certificate. So, there is at least a positive correlation between the scale of an attack and the likelihood that it is detected by these mechanisms.

5.3.3. Forks and Out of Date Peers

For our analysis, we have assumed that the ledger operates in an ideal manner, but this is of course not always the case. In particular, it takes some finite amount of time for a blockchain to converge so that a transaction is accepted by all peers. Furthermore, because of network delays, some peers may be behind in their data. In cryptocurrency applications, two transactions may conflict with each other if they both try to spend the same funds, and hence a transaction which seems early on to be valid to some peers may later on disappear from the record as the blockchain converges. For this reason, it is common to wait for some set period after a transaction is added to the chain to make sure that the chain has well and truly converged.

In our application, the story is different because there is no conflict between transactions. That is to say, whether a particular certificate has been logged or not has no bearing on whether other certificates can be logged. In some cases, the order in which certificates are added may change as the blockchain converges—and this will invalidate membership proofs—but all certificates can be logged. (The one exception to this is that only one revocation is stored per certificate, so if a second revocation for a certificate is issued, then it will not be logged, but this will have little bearing on security, since the effect is the same for either revocation. If storing multiple revocations per certificate is desired, then they can be stored in a Merkle-BST, the root hash of which is stored where the revocation is currently stored). Similarly, if there is a fork in the blockchain, then all the certificates from one fork can be copied over to the other fork. This property actually makes attacks against our application much harder than for cryptocurrencies. If an attacker successfully creates a fork in an attempt to exclude a certificate that was logged, and this fork becomes accepted, then peers could still re-add all the certificates that they know about, including the certificate that was specifically being excluded. Hence, these types of attacks will be largely fruitless. The attacker would need to continuously subvert the consensus process to exclude the certificate any time that it was later added.

Nevertheless, there is still a practical issue with peers being out of sync, which is that clients and servers may not agree on what the most recent root hash is. In these cases, the client would not accept the proof offered by the server. To solve this problem, first suppose that it takes t seconds for the blockchain to converge with high probability, so that all peers agree about all blocks that are t or more seconds old (as measured by the peer who submitted the block). Now, we introduce the following additional behaviours:

- Each peer keeps copies of the trees for all recent blocks back to the most recent block that is more than t seconds old (according to the peer). Note that this can be done efficiently: most nodes in a tree do not change with the addition of a certificate, and hence these nodes can be kept in common for all the blocks' trees if pointers or references are used. Only $O(\log n)$ nodes along the path from the root to the new certificate need to be changed with the addition of a new certificate, and these are what needs to be stored for each block.
- The client will keep a list of root hashes, including all root hashes that are, at most, $2t$ seconds old (according to its peer(s)), plus the next most recent root hash.
- The server requests a proof from a peer from the peer's oldest data structure and offers it to the client

With these behaviours, the proof offered by the server uses the most recent root hash F that is (according to its peer) t or more seconds old, so by assumption, all other peers will have F on their copy of the blockchain, including the peer(s) that the client contacts. If F is at most $2t$ seconds old according to the client's peer(s), or is the next most recent, then the client can check the server's proof.

So now suppose that F is more than $2t$ seconds old according to the client's peer(s) and that there is some other root hash F' that is more recent but still more than $2t$ seconds old according to the client's peer(s), so that F' is the last on client's list. Then, t seconds ago, F' was more than t seconds old and hence known to the server's peer at that time. This means that F' must now be more than t seconds old (according to the server's peer), and more recent than F (because the order of blocks is fixed). This contradicts the choice of F, hence this case cannot happen.

With this mechanism, network delays can be accommodated with high probability. Note that it is also efficient: peers have the highest burden, essentially keeping differences between successive data structures for the last t seconds. Clients must store a list of root hashes from the last $2t$ seconds, and the server has no additional burden. Note that in typical blockchain implementations, new blocks are created on a timescale of minutes (10 minutes per block for Bitcoin, for example [41]) so for reasonable values of t, a very small number of data structures and root hashes need to be stored.

5.4. What Ledger to Use

There are three main possibilities for how append-only ledgers can be implemented: trusted third party, private blockchain, and public blockchain. Since we are aiming for maximum transparency and decentralization, public blockchains are the most reasonable choice. While it is possible to implement a new blockchain for our application, it is likely more convenient to make use of an already-existing blockchain, such as Ethereum [32], which will also allow for additional functionality by providing monetary incentives for certain behaviours.

When speaking of blockchains, the question of which consensus mechanism to use will always come up. Since we are considering public blockchains only, the two main contenders are proof-of-work and proof-of-stake. The former is very costly, but requires little additional consideration other than to make sure that incentives are high enough that honest peers dominate, making the cost of a successful attack high. There are a few ways in which this might be done. One way would be to incorporate a cryptocurrency and reward peers with coins as is typically done in cryptocurrencies. In order for this to survive speculation bubbles, though, the currency needs to be given real value by someone offering some other currency in exchange. This offers one mechanism for paying peers: buy the coins that they mine.

Proof-of-stake is perhaps more nuanced in this application. What exactly is the stake? If a cryptocurrency is incorporated with the ledger, then the answer is obviously coins. Those with the largest stake will be those that have contributed most to the maintenance of the ledger by buying coins, presumably with real money. There are other possibilities, though. For example, a CA's stake might relate to the number of certificates logged that they have signed, or a peer's stake might relate to the number of certificates that they have contributed. However, all of these options leave open the possibility of a de facto centralisation by a small cabal with a large stake. Such a cabal might exclude new CAs by refusing to add their certificates, or refuse to add certificates that cast the cabal in a bad light, such as mistakenly issued certificates by CAs in the cabal.

Given the difficulties of maintaining a separate ledger, such as maintaining a cryptocurrency to incentivise peers, using a pre-existing ledger is likely the most practical option. The existing cryptocurrency can be used in incentive schemes, and the certificate log can be implemented through smart contracts. Since the ledger exists in its own right, there is no need to pay for its maintenance, only its use when new certificates are added. It also makes it much easier to support multiple DPKIT instances.

5.5. Comparison to Other Schemes

In Section 1.2, we mentioned several proposals for improving PKI security, including three broad approaches: logging mechanisms, PKI redesigns utilising blockchains, and schemes that augment traditional PKI with blockchains. Here, we compare the security properties of these schemes to our own.

First, consider PKI transparency systems, such as certificate transparency and the SSL observatory. Our system is similar to these systems, in that they are all backwards-compatible with the existing PKI infrastructure, leading to easier adoption. As well, they all keep the existing system of semi-trusted certificate authorities for real-world identification verification. However, our proposal is fully decentralised once a certificate has been issued, reducing the number of single points of failure (of trust as well as functionality). Furthermore, our proposal supports certificate revocation, which is absent from CT.

On the other hand, fully decentralised PKI redesigns are not backwards compatible with existing the PKI, and hence this is a barrier to adoption. Furthermore, because they are fully decentralised, there is no means of checking online identities against real world identities. Feng et al.'s proposal [25] deals with this problem by introducing supervisory nodes, which play an analogous role for CAs, resulting in a mixed centralised/decentralised system that is in some ways similar to ours, but not backwards-compatible with existing systems.

CertLedger [26] is very similar to our scheme in many ways and includes more functionality, but at the cost of increased centralisation. The CertLedger board, for example, has control over which CAs are considered trusted, so this choice is taken away from clients. In particular, CertLedger offers no additional security to a client using a CA that has not been endorsed by the CertLedger board. Revocations in CertLedger are handled through a signed request to CertLedger rather than through a revocation certificate. This means that additional information needs to be handled through the ledger, namely the ownership of domains, as domain owners can request a revocation. Certificates are also added by domain owners only, meaning that problematic certificates are not logged. Hence, CertLedger is much less transparent than our scheme, or other schemes like CT. These choices mean that CertLedger assumes a large amount of control over PKI processes, sacrificing flexibility and adoptability. Our scheme, by contrast, still has value as a certificate store that clients can consult when making security decisions, even if domain owners and CAs are not involved.

Like our scheme and CT, CTB [27] aims primarily to provide a public certificate store. However, it is more centralised that our scheme, using a permissioned blockchain operated by CAs. Having the CAs as the central authorities for the ledger introduces the possibility that they will attempt to block certificates from being logged if they would damage their reputation, although competing CAs might be able to publish such certificates anyway, depending on the specifics of the consensus mechanism. As well, the participation of CAs is required for adoption. CTB does not provide a privacy-preserving method of checking whether a certificate has been logged as our scheme does. Instead, clients query the ledger directly. Revocations are supported by recording the revocation status of each certificate.

Blockchain-based certificate transparency [28] is very similar to our scheme, with privacy preserving proofs of membership for clients, and revocations. However, it only logs certificates that have been published by the domain owner through the use of publishing keys. This considerably reduces the transparency in much the same way as CertLedger: problematic certificates are not logged, only valid certificates, and no certificates are logged for a domain if the owner does not participate. Domain owners' publishing keys are valid once certified by a group of trusted certifiers. This re-introduces some degree of centralisation, although certifiers are chosen based on their historical behaviour on the blockchain, rather than by a centralised authority.

CA proxies [29] have a different goal from the previous examples, instead focusing on making the processes of issuing and revoking more transparent certificates by posting certificate signing requests publicly along with issued certificates and revocations. These goals are largely orthogonal to ours. For example, CA proxies have no mechanism to track problematic certificates, only certificates that the CA wishes to be published. Nor is there a mechanism for clients to efficiently discover whether a certificate has been published or revoked through CA proxy.

The proposal by Yakubov et al. [30] uses a blockchain to store certificates and revocations, much like our scheme. However, it is not clear what functionality the proposal offers to clients. Certificates are stored in an array, meaning that efficient searching or proofs of membership are unlikely. Validation of a certificate is made via a call to a smart contract on the blockchain (which does not change the blockchain and hence incurs no cost to run), and hence is unlikely to be privacy-preserving. Validation does not appear to include checks for revocations, only verification of the certificate and intermediate CA certificates on the path to the relevant root CA.

Since DPKIT also provides a mechanism for discovering if a certificate is revoked, we can reasonably compare it to other revocation mechanisms. Certificate revocation lists are problematic as single points of failure, vulnerable to denial of service attacks, and are also inefficient, requiring clients to frequently download lists from multiple sources. DPKIT, on the other hand, is efficient, since the client only receives information pertaining to a particular certificate, and is decentralised leading to greater robustness against denial of service attacks. OSCP is an improvement on CRLs, but introduces privacy concerns, since the client must reveal which certificates it is interested in a to third party. In DPKIT, the client receives the proof from the server and does not need to contain a third party, protecting privacy. OSCP stapling works similarly, so that the client receives the OSCP response from the server, not a third party. However, some types of attacks may not be detected through OSCP stapling that would be detected by using DPKIT. In particular, if a CA private key is compromised, then this could be used to create a valid OSCP response that would be accepted by the client. With DPKIT, the existence of a revocation cannot be hidden from the client, even with a compromised CA private key, since its existence in the store is independent of any CA signatures.

6. Conclusions

We have proposed, modelled, and instantiated a new decentralised approach to certificate transparency based on generic blockchains augmented with efficient data structures. It is compatible with existing PKI, as customary entities are kept to manage basic functionalities. DPKIT is expected to be highly effective at catching and deterring trust abuses by malicious certificate authorities.

Public key infrastructures enable users to look up and verify each other's public keys based on identities. An identity, to the greatest possible extent, should only be issued, updated or revoked after the permission from the owner of that identity. The traditional PKI model does not effectively prevent one user from registering another's already registered public key as their own. Lately, various approaches [20,23] have been proposed to guarantee strong identity retention properties which do not rely on a trusted third party. As for now, we have not incorporated an identity retention mechanism in DPKIT, which could be obtained by us having certificate updates (issuance and/revocation) cross-signed by the domain owner's signature. Certificate mis-issuance problem can also be solved by Certification Authority Authorization [42], which is an Internet security policy mechanism that allows a DNS domain name holder to specify one or more CAs authorized to issue certificates for that domain.

Constructing a blockchain-based PKI is a feasible alternative method for the mainstream issues related to conventional and log-based approaches. We present DPKIT as a platform that offers more transparency and decentralization than existing PKI. Based on the security notions that we formalized, we have shown that our scheme prevents the misbehaviour of entities involved. Having certificate revocation built in our system is a benefit that is not provided by current PKI schemes. As revocation information is included in the proof of certificate itself, it mitigates the need for centrally administered OCSP servers run by CAs.

Author Contributions: Conceptualization: X.B., U.H., M.M. and D.S.; Formal analysis: U.H.; Supervision: X.B., M.C. and D.S.; Writing—original draft: U.H.; Writing—review and editing: M.M. All authors have contributed, read and agreed to the published version of the manuscript.

Funding: X.B. acknowledges support as a Future Fellow from the Australian Research Council, grant number FT140101145. The APC was funded by X.B.'s ARC Future Fellowship grant FT140101145.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All of the reported data is included in the manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Construction of Merkel-BST

Definition A1 (associative array with Membership Proofs). *An associative array with membership proofs consists of the following algorithms, where t is a data structure representing the overall state of the system. The first few algorithms are used to manage entries in the associative array.*

- $\mathrm{Init}_M(\mathrm{FP}, \mathrm{H}) \to t$*: deterministic algorithm that give a value fingerprinting hash function* FP *and a hash function* H*, outputs an initial data structure t.*
- $\mathrm{Insert}_M(t, k, v) \to e$ *: deterministic algorithm that takes a key-value entry* (k, v)*, a data structure t, and returns* 1 *if successful or* 0 *on error.*
- $\mathrm{Update}_M(t, k, v) \to e$ *: deterministic algorithm that takes a key-value entry* (k, v)*, a data structure t, and returns* 1 *if successful or* 0 *on error.*
- $\mathrm{Search}_M(t, k) \to v$ *or* $\perp$*: A deterministic algorithm that takes an entry key k and a data structure t, and outputs a value v or* $\perp$*.*

  The next algorithms are used to test various properties of the associative array:
- $\mathrm{GetFingerprint}_M(t) \to F$*: deterministic algorithm that takes a data structure t and outputs a fingerprint F, representing the complete list of entries.*
- $\mathrm{GetAll}_M(t) \to (E)$*: deterministic algorithm that takes a data structure t and outputs an ordered list of entries E.*

  The last few algorithms generate and verify membership proofs:
- $\mathrm{GetProof}_M(t, k) \to P$ *or* $\perp$*: deterministic algorithm that takes as inputs an entry key k and a data structure t and outputs a membership proof P or* $\perp$ *if the key is not present.*
- $\mathrm{VerifyProof}_M(P, F, kh, vh, H) \to \{0, 1\}$*: A deterministic algorithm that takes as inputs an entry key hash kh, a value fingerprint vh, a data-structure fingerprint F, a membership proof P, and a hash function H and outputs a bit* $b \in \{0, 1\}$*.*

Now, we define the correctness properties of our associative array with membership proofs.

Definition A2. *(Correctness of an associative array with membership proofs) An associative array with membership proofs is correct if,*

1. *If* $\mathrm{Insert}_M(t, k, v)$ *is called,* $\mathrm{Insert}_M(t, k, v')$ *has not been called previously, and* $\mathrm{Update}_M(t, k, v')$ *has not been called since, then any subsequent call* $\mathrm{GetAll}_M(t)$ *will return a list that contains the pair* (k, v)*.*
2. *If* $\mathrm{Insert}_M(t, k, v)$ *is called,* $\mathrm{Update}_M(t, k, v')$ *is called, and* $\mathrm{Update}_M(t, k, v'')$ *has not been called since, then any subsequent call* $\mathrm{GetAll}_M(t)$ *will return a list that contains the pair* (k, v')*.*
3. *Any call* $\mathrm{Search}_M(t, k)$ *will return the value v corresponding to the first call* $\mathrm{Insert}_M(t, k, v)$ *or the most recent call to* $\mathrm{Update}_M(t, k, v)$ *after a call to* Insert_M *with matching key k, or an error symbol* $\perp$ *if no such call had been made.*
4. *Under the same conditions as above, it is always the case that* $\mathrm{VerifyProof}_M(\mathrm{GetProof}(t, k), \mathrm{GetFingerprint}_M(t)) = 1$.

Definition A3. *(Security of proofs of membership for associative array with membership proofs)*
We say that an associative array with membership proofs scheme θ *using hash function* H *and fingerprint function* FP *is* ϵ*-secure with respect to* $\mathbb{A}$*, for all* $\mathcal{A} \in \mathbb{A}$*,*

$$P(\mathrm{Exp}_{\theta}^{\textsf{prooof-unforgability}}(\mathcal{A}, \mathrm{H}, \mathrm{FP}) = 1) < \epsilon$$

where $\mathrm{Exp}^{\textsf{prooof-unforgability}}$ *is given in Figure A1.*

```
Exp_θ^{prooof-unforgability}(A, H, FP):
1: $t \leftarrow \mathrm{Init}_\theta(\mathrm{FP}_M, \mathrm{H})$
2: $(P, k, v) \leftarrow \mathcal{A}()^{\theta*(t,\cdot)}$
3: if $(k, v) \notin \mathrm{GetAll}_\theta(t)$, return 0
4: return $\mathrm{VerifyProof}_\theta(P, t.root.hash, \mathrm{H}(k), \mathrm{FP}(v), \mathrm{H})$
```

Figure A1. Experiment for security of membership proofs.

```
Init_M(FP, H) → t:
1: $t \leftarrow (\,)$
2: $t.root \leftarrow \emptyset$
3: $t.fp \leftarrow \mathrm{FP}$
4: $t.hash \leftarrow \mathrm{H}$
5: return $t$

CreateNode_M(t, k, v, p) → n:
1: $n \leftarrow (\,)$
2: $n.k \leftarrow k$, $n.v \leftarrow v$, $n.parent \leftarrow p$
3: $n.right \leftarrow \emptyset$, $n.left \leftarrow \emptyset$
4: $n.hash \leftarrow \mathrm{NodeHash}_M(t, n)$
5: return $n$

Insert_M(t, k, v) → e:
1: if $(t.root = \emptyset)$
2:    $t.root \leftarrow \mathrm{CreateNode}_M(t, k, v, \emptyset)$
3:    return 1
4: else
5:    return $\mathrm{InsertNode}_M(t.root, k, v)$

InsertNode_M(n, k, v) → e:
1: if $(k = n.k)$ return 0
2: if $(k < n.k)$
3:    if $(n.left = \emptyset)$
4:       $n.left \leftarrow \mathrm{CreateNode}_M(t, k, v, n)$
5:       $r \leftarrow 1$
6:    else
7:       $r \leftarrow \mathrm{Insert}_M(n.left, k, v)$
8: else
9:    if $(n.right = \emptyset)$
10:      $n.right \leftarrow \mathrm{CreateNode}_M(t, k, v, n)$
11:      $r \leftarrow 1$
12:   else
13:      $r \leftarrow \mathrm{Insert}_M(n.right, k, v)$
14: if $(r = 0)$ return 0
15: $n.hash \leftarrow \mathrm{NodeHash}_M(t, n)$
16: return 1

Search_M(t, k) → v:
1: $n \leftarrow \mathrm{SearchNode}_M(t.root, k)$
2: if $(n = \perp)$ return $\perp$
3: return $n.v$

SearchNode_M(n, k) → n':
1: if $(n = \emptyset)$ return $\perp$
2: if $(k = n.k)$ return $n$
3: if $(k < n.k)$
4:    return $\mathrm{SearchNode}_M(n.left, k)$
5: return $\mathrm{SearchNode}_M(n.right, k)$

Update_M(t, k, v) → e:
1: $n \leftarrow \mathrm{SearchNode}_M(t.root, k)$
2: if $(n = \perp)$ return 0
3: $n.v \leftarrow v$
4: $n.hash \leftarrow \mathrm{NodeHash}_M(t, n)$
5: while $(n.parent \neq \emptyset)$
6:    $n \leftarrow n.parent$
7:    $n.hash \leftarrow \mathrm{NodeHash}_M(t, n)$
8: return 1

NodeHash_M(t, n) → h:
1: return $\mathrm{NodeHashN}_M(t.hash(n.k),$
   $t.fp(n.v), n.left, n.right, t.hash)$

NodeHashN_M(kh, vh, l, r, H) → h:
1: if $(l = \emptyset) \wedge (r = \emptyset)$
2:    return $H(1\|kh\|vh)$
3: if $(r = \emptyset)$
4:    $h_3 = l.hash$
5:    return $H(2\|kh\|vh\|h_3)$
6: if $(l = \emptyset)$
7:    $h_3 = r.hash$
8:    return $H(3\|kh\|vh\|h_3)$
9: $h_3 = l.hash, h_4 = r.hash$
10: return $H(4\|kh\|vh\|h_3\|h_4)$

GetFingerprint_M(t) → F:
1: $F \leftarrow t.root.hash$
2: return $F$

GetAll_M(t) → E:
1: if $(t.root = \emptyset)$, return $\perp$
2: return $\mathrm{GetAllNode}_M(t.root)$

GetAllNode_M(n) → E:
1: if $(n.left \neq \emptyset)$
2:    $E \leftarrow \mathrm{GetAllNode}_M(n.left)$
3: else
4:    $E \leftarrow [\,]$
5: $E.append((n.k, n.v))$
6: if $(n.right \neq \emptyset)$
7:    $E \leftarrow E \,||\, \mathrm{GetAllNode}_M(n.right)$
8: return $E$
```

Figure A2. Construction of Merkle BST. The main data structure is $t = (root : r, fp : f, hash : h)$ where r is a node, f is a fingerprint function for the values, and h is a hash function. Nodes are $n = (k : key, v : value, parent : p, left : l, right, r)$ where *key* and *value* are types defined by the application, and p, l, r are nodes.

With this definition is place, we can formally state Lemma 1

Lemma A1 (Unforgability of Merkle-BST memberships proofs). *If* H *and* FP *are ϵ-collision resistant with respect to adversaries $\mathbb{A}$, then Merkle-BST instantiated with* H *and* FP *is ϵ-*secure *with respect to $\mathbb{A}$.*

```
GetProof_M(t, k) → P:
1:  return GetProofN_M(t, t.root, k)

GetProofN_M(t, n, k) → P:
1:  if (k = n.k)
2:      lh ← n.left ≠ ∅ ? n.left.hash : ∅
3:      rh ← n.right ≠ ∅ ? n.right.hash : ∅
4:      return [(∗, ∗, lh, rh)]
5:  if (k < n.k)
6:      if (n.left = ∅) return ⊥
7:      rh ← n.right ≠ ∅ ? n.right.hash : ∅
8:      P ← GetProofN_M(t, n.left, k)
9:      if (P = ⊥) return ⊥
10:     return P || [(t.hash(n.k), t.fp(n.v), ∗, rh)]
11: if (n.right = ∅) return ⊥
12: lh ← n.left ≠ ∅ ? n.left.hash : ∅
13: P ← GetProofN_M(t, n.right, k)
14: if (P = ⊥) return ⊥
15: return P || [(t.hash(n.k), t.fp(n.v), lh, ∗)]
```

Figure A3. Algorithm GetProof of Merkel-BST. A proof is a list of tuples (kh, vh, lh, rh) where kh is the hash of a key, vh is the fingerprint of a value, lh is the hash of a node, and rh is the hash of a node. $*$ is used where a hash must be calculated by the verifier from available information. $\emptyset$ is used in place of lh or rh to indicate that a child is absent.

```
VerifyProof_M(P, F, kh, vh, H) → b:
1:  if (VerifyProofN_M(P, kh, vh, H) = F) return 1
2:  return 0

VerifyProofN_M(P, kh, vh, H) → h:
1:  (kh′, vh′, lh, rh) ← P[|P| − 1]
2:  if (|P| = 1)
3:      if (lh = ∗ ∨ rh = ∗ ∨ kh′ ≠ ∗ ∨ vh′ ≠ ∗) return ⊥
4:      return NodeHashN_M(kh, vh, lh, rh, H)
5:  if (lh = ∗ ∧ rh = ∗) return ⊥
6:  if (lh ≠ ∗ ∧ rh ≠ ∗) return ⊥
7:  if (lh = ∗)
8:      lh ← VerifyProofN_M(P[0 : |P| − 1], kh′, vh′, H)
9:  else
10:     rh ← VerifyProofN_M(P[0 : |P| − 1], kh′, vh′, H)
11: return NodeHashN_M(kh′, vh′, lh, rh, H)
```

Figure A4. Algorithm VerifyProof of Merkle BST.

The proof of this fact is similar to that for Merkle trees, and hence is left as an exercise.

References

1. Dierks, T.; Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.1. Internet Engineering Task Force (IETF). 2006, RFC 4346. Available online: https://tools.ietf.org/id/draft-ietf-tls-oldversions-deprecate-02.html (accessed on 21 March 2021).
2. Cooper, D.; Santesson, S.; Farrell, S.; Boeyen, S.; Housley, R.; Polk, W. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), 2008. Updated by RFC 6818. Available online: https://datatracker.ietf.org/doc/rfc5280 (accessed on 21 March 2021).

3. Fox IT. Black Tulip: Report of the Investigation into the DigiNotar Certificate Authority Breach. 2012. Available online: http://www.rijksoverheid.nl/bestanden/documenten-en-publicaties/rapporten/2012/08/13/black-tulip-update/black-tulip-update.pdf (accessed on 21 March 2021).
4. Comodo Group. Comodo Fraud Incident: Update 31-Mar-2011. 2011. Available online: https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html (accessed on 21 March 2021).
5. Ducklin, P. The TURKTRUST SSL Certificate Fiasco—What really Happened, and What Happens Next? Available online: https://nakedsecurity.sophos.com/2013/01/08/the-turktrust-ssl-certificate-fiasco-what-happened-and-what-happens-next/ (accessed on 21 March 2021).
6. Somogyi, S.; Eijdenberg, A. Improved Digital Certificate Security. 2015. Available online: http://googleonlinesecurity.blogspot.de/2015/09/improved-digital-certificate-security.html (accessed on 21 March 2021).
7. Solo, D.; Housley, R.; Ford, W. Internet X. 509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Internet Engineering Task Force (IETF) 2002, RFC 3280. Available online: https://www.ietf.org/rfc/rfc3280.txt (accessed on 21 March 2021).
8. Eastlake, D. Transport Layer Security (TLS) Extensions: Extension Definitions. Internet Engineering Task Force (IETF) January, 2011, RFC 6066. Available online: https://datatracker.ietf.org/doc/html/rfc6066 (accessed on 21 March 2021).
9. Galperin, S.; Santesson, S.; Myers, M.; Malpani, A.; Adams, C.X. 509 Internet Public Key Infrastructure Online Certificate Status Protocol-OCSP. Internet Engineering Task Force (IETF) 2013, RFC 6960. Available online: https://datatracker.ietf.org/doc/html/rfc2560 (accessed on 21 March 2021).
10. Boneh, D.; Franklin, M. Identity-Based Encryption from the Weil Pairing. In *Advances in Cryptology—CRYPTO 2001*; Springer: Berlin/Heidelberg, Germany, 2001; Volume 2139, pp. 213–229.
11. Eckersley, P.; Burns, J. The EFF SSL Observatory. 2010. Available online: https://www.eff.org/observatory (accessed on 21 March 2021).
12. Laurie, B.; Langley, A.; Kasper, E. Certificate Transparency. RFC 6962 (Experimental), 2013. Available online: https://datatracker.ietf.org/doc/html/rfc6962 (accessed on 21 March 2021).
13. Laurie, B. Certificate Transparency. *ACM Queue Secur.* **2014**, *12*, 10. Available online: https://queue.acm.org/detail.cfm?id=2668154 (accessed on 21 March 2021). [CrossRef]
14. Dowling, B.; Günther, F.; Herath, U.; Stebila, D. Secure logging schemes and Certificate Transparency. In *ESORICS*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 140–158.
15. Basin, D.A.; Cremers, C.J.F.; Kim, T.H.; Perrig, A.; Sasse, R.; Szalachowski, P. ARPKI: Attack Resilient Public-Key Infrastructure. In *ACM CCS 2014*; Ahn, G., Yung, M., Li, N., Eds.; ACM: New York, NY, USA , 2014; pp. 382–393. [CrossRef]
16. Kim, T.H.; Huang, L.; Perrig, A.; Jackson, C.; Gligor, V.D. Accountable key infrastructure (AKI): A proposal for a public-key validation infrastructure. In *WWW 2013*; Schwabe, D., Almeida, V.A.F., Glaser, H., Baeza-Yates, R., Moon, S.B., Eds.; International World Wide Web Conferences Steering Committee/ACM: New York, NY, USA , 2013; pp. 679–690.
17. Yu, J.; Cheval, V.; Ryan, M. DTKI: A new formalized PKI with verifiable trusted parties. *Comput. J.* **2016**, *59*, 1695–1713. [CrossRef]
18. Brunner, C.; Knirsch, F.; Unterweger, A.; Engel, D. A Comparison of Blockchain-based PKI Implementations. In *ICISSP*; Science and Technology Publications: Setúbal, Portugal, 2020 ; pp. 333–340.
19. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. Available online: bitcoin.org (accessed on 21 March 2021).
20. Fromknecht, C.; Velicanu, D.; Yakoubov, S. A Decentralized Public Key Infrastructure with Identity Retention. *IACR Cryptol. ePrint Arch.* **2014**, *2014*, 803.
21. Leiding, B.; Cap, C.H.; Mundt, T.; Rashidibajgan, S. Authcoin: Validation and Authentication in Decentralized Networks. *arXiv* **2016**, arXiv:1609.04955. Available online: http://xxx.lanl.gov/abs/1609.04955 (accessed on 21 March 2021).
22. Fredriksson, B. A Distributed Public Key Infrastructure for the Web Backed by a Blockchain, 2017. Available online: http://www.diva-portal.org/smash/get/diva2:1121040/FULLTEXT01.pdf (accessed on 21 March 2021).
23. Qin, B.; Huang, J.; Wang, Q.; Luo, X.; Liang, B.; Shi, W. Cecoin: A decentralized PKI mitigating MitM attacks. *FGCS* **2017**, *107*, 805–815. [CrossRef]
24. Project, T.E. Modified Merkle Patricia Trie Specification. Available online: https://github.com/ethereum/wiki/wiki/Patricia-Tree (accessed on 21 March 2021).
25. Feng, T.; Chen, W.; Zhang, D.; Liu, C. One-Stop Efficient PKI Authentication Service Model Based on Blockchain. In *Blockchain Technology and Application*; Si, X., Jin, H., Sun, Y., Zhu, J., Zhu, L., Song, X., Lu, Z., Eds.; Springer: Singapore, 2020; pp. 31–47. [CrossRef]
26. Kubilay, M.Y.; Kiraz, M.S.; Mantar, H.A. CertLedger: A new PKI model with Certificate Transparency based on blockchain. *Comput. Secur.* **2019**, *85*, 333–352. [CrossRef]
27. Madala, D.S.V.; Jhanwar, M.P.; Chattopadhyay, A. Certificate Transparency Using Blockchain. In Proceedings of the 2018 IEEE International Conference on Data Mining Workshops (ICDMW), Singapore, 17–20 November 2018; pp. 71–80. [CrossRef]
28. Wang, Z.; Lin, J.; Cai, Q.; Wang, Q.; Zha, D.; Jing, J. Blockchain-based Certificate Transparency and Revocation Transparency. *IEEE Trans. Dependable Secur. Comput.* **2020**, 1. [CrossRef]
29. Zhao, J.; Lin, Z.; Huang, X.; Zhang, Y.; Xiang, S. TrustCA: Achieving Certificate Transparency through Smart Contract in Blockchain Platforms. In Proceedings of the 2020 International Conference on High Performance Big Data and Intelligent Systems (HPBD IS), Shenzhen, China, 23 May 2020; pp. 1–6. [CrossRef]

30. Yakubov, A.; Shbair, W.M.; Wallbom, A.; Sanda, D.; State, R. A blockchain-based PKI management framework. In Proceedings of the NOMS 2018—2018 IEEE/IFIP Network Operations and Management Symposium, Taipei, Taiwan, 23–27 April 2018; pp. 1–6. [CrossRef]
31. Kent, S. Attack Model and Threat for Certificate Transparency. Internet Engineering Task Force (IETF). 2015. Internet Draft. Available online: https://datatracker.ietf.org/doc/html/draft-ietf-trans-threat-analysis (accessed on 21 March 2021).
32. Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Proj. Yellow Pap.* **2014**, *151*, 1–32.
33. Merkle, R.C. A Certified Digital Signature. In *CRYPTO'89*; Brassard, G., Ed.; Springer: Berlin/Heidelberg, Germany, 1990; Volume 435, pp. 218–238. [CrossRef]
34. Merkle, R.C. Secrecy, Authentication, and Public Key Systems. 1979. Available online: https://www.merkle.com/papers/Thesis1979.pdf (accessed on 21 March 2021).
35. Cormen, T.H. *Introduction to Algorithms*, 3rd ed.; MIT Press: Cambridge, MA, USA, 2009.
36. Barber, S.; Boyen, X.; Shi, E.; Uzun, E. Bitter to better—how to make bitcoin a better currency. In *FC'12*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 399–414.
37. Eyal, I.; Sirer, E.G. Majority is not enough: Bitcoin mining is vulnerable. In *FC'14*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 436–454.
38. Li, X.; Jiang, P.; Chen, T.; Luo, X.; Wen, Q. A survey on the security of blockchain systems. *Future Gener. Comput. Syst.* **2020**, *107*, 841–853. [CrossRef]
39. Conti, M.; Sandeep Kumar, E.; Lal, C.; Ruj, S. A Survey on Security and Privacy Issues of Bitcoin. *IEEE Commun. Surv. Tutor.* **2018**, *20*, 3416–3452. [CrossRef]
40. Feng, Q.; He, D.; Zeadally, S.; Khan, M.K.; Kumar, N. A survey on privacy protection in blockchain system. *J. Netw. Comput. Appl.* **2019**, *126*, 45–58. [CrossRef]
41. Bitcoin difficulty. Available online: https://en.bitcoin.it/wiki/Difficulty (accessed on 8 March 2019).
42. Hallam-Baker, P.; Stradling, R. DNS Certification Authority Authorization (CAA) Resource Record. Technical Report, 2013. Available online: https://datatracker.ietf.org/doc/rfc6844/?include_text=1 (accessed on 21 March 2021).