Zentrum für sichere Informationstechnologie – Austria
Secure Information Technology Center – Austria

A-SIT

Seidlgasse 22 / 9, 1030 Wien
Tel.: (+43 1) 503 19 63–0
Fax: (+43 1) 503 19 63–66

Inffeldgasse 16a, 8010 Graz
Tel.: (+43 316) 873-5514
Fax: (+43 316) 873-5520

http://www.a-sit.at
E-Mail: office@a-sit.at

DVR: 1035461
ZVR: 948166612
UID: ATU60778947

TECHNICAL ANALYSIS OF THE GENERAL TRANSPARENCY PARADIGM

Version 1.0, 25.03.2020
Edona Fasllija edona.fasllija@iaik.tugraz.at

Abstract:
This project analyzed alternative ways to enable transparency for stakeholders in a trust service-based ecosystem, besides traditional third-party certification. One of the most popular trends in this area is Certificate Transparency (CT) initiated by Google [1]. The underlying principles of CT or other transparency-related efforts (such as Revocation Transparency [2], Key Transparency [3], etc.) are not specific to a domain and can be used to add transparency in general to a trust model. This Transparency paradigm allows for a more decentralized way of generating trust. This project analyzed the data structures and their applications that would enable adding transparency to a trust model and investigated further trust ecosystems that this model can benefit.

Zusammenfassung:
Dieses Projekt hat analysiert, inwieweit die von Gogle eingeführte Certificate Transparency (CT) als Ansatz zur Verhinderung von Missbrauch auf andere Vertrauensdienste ausweitbar ist. Die in CT (oder verwandten Themen wie Key Transparency oder Revocation Transparency) angewandten Prinzipien lassen sich potentiell generalisieren, um dezentrale Vertrauensmodelle aufzubauen. Es wurden im Projekt Datenstrukturen und Anwendungen dahingehend untersucht, inwieweit diese für solch generalisierte Transparenzanasätze geeignet sind.

Table of Contents

1. Introduction

Maintaining audit trails, or chronological records of events is crucial to many organizations and authorities to provide proof of compliance and operational integrity. In these trust models, several ecosystems need to trust the authorities to preserve the integrity of the stored data. However, many database or log systems may be vulnerable to tampering, either due to their deployment to untrusted servers or potential malicious insider attacks.
The *Transparency* paradigm refers to several data structures and their applications that allow adding transparency to such trust models. It enables the establishment of trust relationships by leveraging verifiable cryptographic proofs. It provides transparency by making *evident* any non-authorized change on the data, even when deployed into non-trusted servers.
In this project, we analyze the state of the art approaches that have been previously explored and employed to build a system that can make evident any non-authorized change to stored data.

1. Authenticated Data Structures

The target data structure of the trust model is a *continuous sequence of logs* (for example, access logs, event logs, transaction logs, etc.) stored on one or multiple logs. Clients need to trust the log to correctly implement and operate the log, by not leaving out entries, and not deleting or tampering log entries that have been previously entered.
For such a system, we can identify three principal agents: the ***source***, which creates the data records, a ***logger*** that instantiates and maintains an append-only log, and one or multiple ***auditors***.

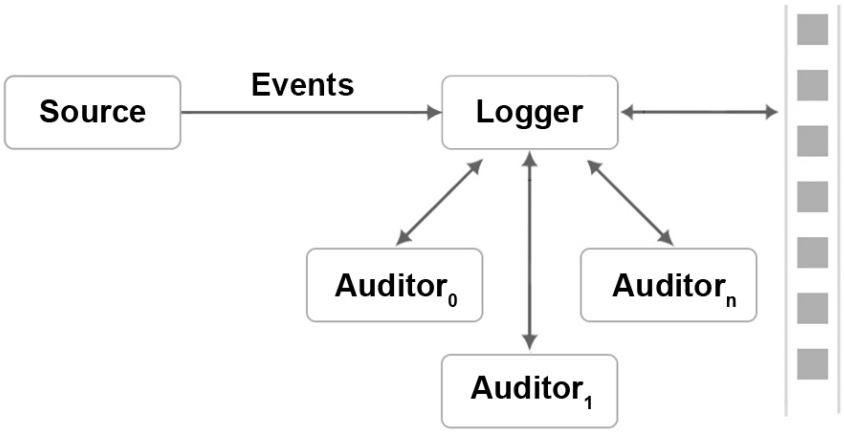

Figure 1 Trust Model

The log is required to have three main properties:

i. It must be possible to efficiently construct a proof for a specific record R in a log, that allows the client to verify that R exists in the log.
ii. The client can efficiently verify that an earlier log is a prefix of the current log.
iii. The auditor can efficiently iterate over the entries of the log.

The ***logger*** is never assumed to be trusted in generating results in response to queries. This is why continuous auditing of the log is crucial to ensure its correctness. Unless audited by one or more trusted ***auditors***, tampering of the data records cannot be detected. The process of auditing involves two main roles: namely the ***prover*** and ***verifier*** role. The *prover* role (played by the logger) *stores* data and *answers queries* about it.
Along with these responses, the prover also returns **proof of validity** of the answer. The verifier role is played by the auditor. The auditor **makes queries** to the logger and **verifies** the results by leveraging the proof returned.

Such tamper-evident logs, also called ***transparent logs***, implement a special kind of data structure called *authenticated data structure* (ADS) [6]. These special data structures allow their operations in response to user queries to be carried out by untrusted provers. ADSs, therefore, are the main element that allows moving data to *untrusted servers* without losing *data integrity*.
The two main cryptographic primitives that transform regular data structures into authenticated ones are *cryptographic hash functions* and *digital signatures*. Hash functions comprise secure compression mechanisms that map a large space of possible records to a smaller space of digests with computationally hard-to-find collisions. On the other hand, digital signatures provide authenticity, non-repudiation, and integrity of records.
In the next section, we analyze alternative data structures that have been proposed for the purpose of designing and implementing a transparent log.

1.1. Hash chains

One of the first approaches proposed as ADSs were hash chains [9]. As the name implies, a hash chain is a chain of commitments, where each commitment is basically a digest that snapshots the entire log up to that point. Hash chains allow for the establishment of a provable order between entries. Furthermore, this technique ensures that any commitment to a given state of the log is implicitly a commitment to all prior states. Therefore, any attempt to remove or alter some log entries will subsequently invalidate the hash chain.

Figure 2 Hash Chains

A hash chain is required to be able to produce three kinds of proofs, namely ***proof of inclusion***, ***proof of consistency***, and ***proof of deletion***, to prove that an entry has been included in a chain and that it has not been modified inconsistently.
One disadvantage of hash chains is that they present linear space and time behavior (*O(n)*) because the auditor needs to examine every intermediate event between snapshots for a simple membership query.

1.2. Merkle Trees

Merkle trees are intuitively simple data structures that allow the logger to commit to a whole set of entries instead of single entries, and then give an inclusion proof of the entry in a log without revealing the whole log.
A Merkle tree is basically a *binary* tree where each node has an *associated cryptographic hash*. In a Merkle tree, *leaf nodes* are associated with the hash of the data records. Hence, leaf nodes contain the actual hash of the data as their *label*. *Parent nodes* are labeled by computing a hash function on their children's hashes concatenated left to right. The label of a parent node can hence be calculated as $\mathrm{H}(\mathrm{L}(\mathrm{cleft})||\mathrm{L}(\mathrm{cright}))$, where H denotes the collision-resistant hash function,
L denotes the label of its left (c_{left}) and right child(c_{right}), respectively. The resulting root hash then serves as a *tamper-evident summary* (commitment) of all log contents (Figure 3).

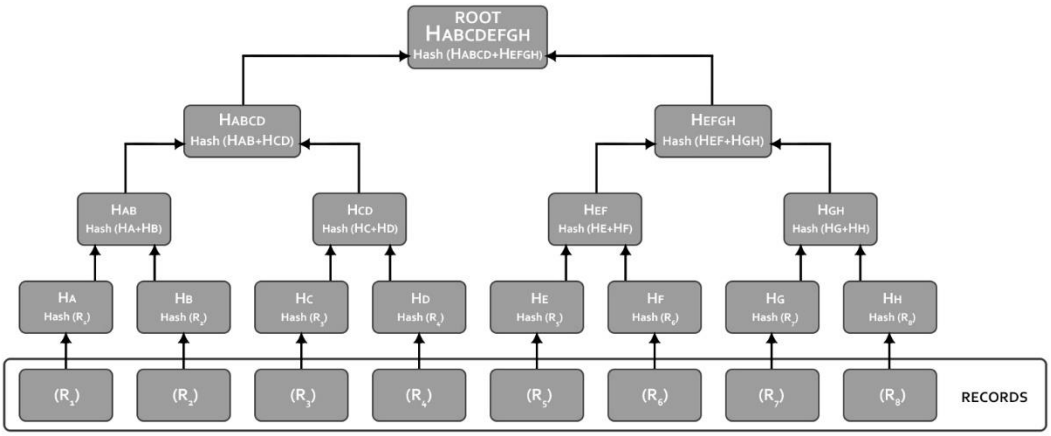

Figure 3 Merkle Trees [11]

Due to their structure, Merkle trees allow for the efficient generation of proof that a particular data is included in the tree. Such proofs are called *inclusion* or *membership* proofs. To prove that a particular record R is included in the log, we start at the leaf node that is associated with the hash of the record R and walk up the tree while redoing the concatenation and hashing at each step. For example, in the image above, to prove the inclusion of R_3 in the log, the proof would consist of H_D, H_{AB}, and H_{EFGH}, because we encounter these hashes on the other side of the branch when walking up the tree (Figure 4).

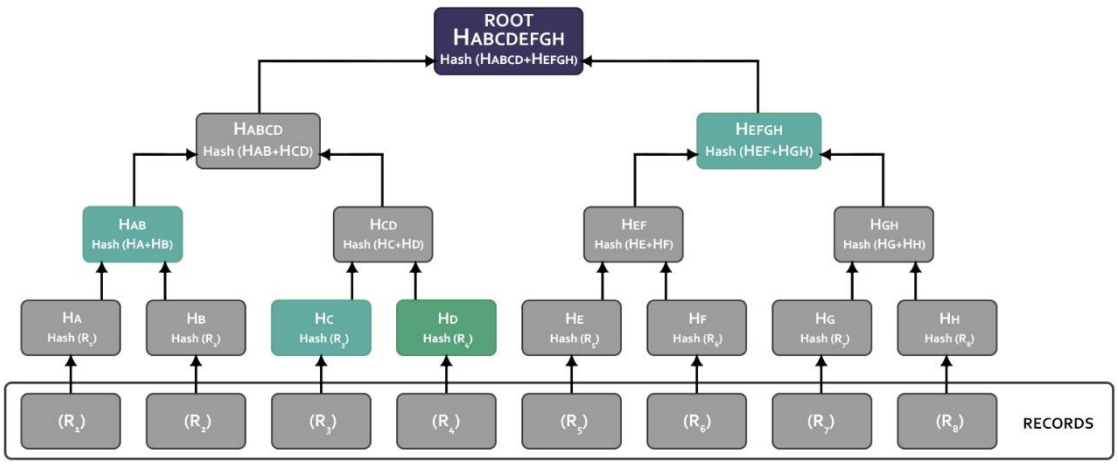

Figure 4 Merkle Audit Path for Record R_3

One can immediately notice that the information contained in an *inclusion proof* is sufficient to recompute the label of the root node, also called the *root hash*. All the nodes required to compute the root hash for the tree comprise the *audit path* for a leaf, also known as *Merkle audit paths*. Since the minimum audit path includes one node for every layer of the tree, the size of the proof becomes logarithmic (*O(log n)*). The structure of the audit path (pruned tree) also offers efficient query and verification times of *O(log n)*.
In order to verify an audit path, all hashes, bottom-up, from the leaf towards the root node need to be recomputed. Once the root hash is recomputed, it is then compared with the original root hash of the Merkle tree—the commitment. The proof is believed to be valid if both match, provided that a trustworthy root hash is available for comparison.

In order to obtain such a trustworthy root hash, Merkle Trees can have a trusted party sign the commitments using digital signatures.

Besides offering efficient membership proofs, Merkle Trees also use only *linear* space, given the fact that they store data only in the leaves.
However, Merkle trees also have several drawbacks. Update scenarios are an important limitation of Merkle Trees: when an update operation is performed, it is not possible to reconstruct past versions of that tree.
Furthermore, Merkle trees cannot satisfy the requirements of dynamic data structures. A fundamental requirement of such dynamic scenarios is that the log is *persistent*. Persistent ADSs are ADSs able to capture their history of changes.
In the following section, we describe a persistent version of Merkle trees, namely History Trees.

1.3. History Trees

If the records of a Merkle tree are filled left-to-right at each level, while keeping the insertion order, it would enable us to reproduce the behavior of a log. Such a binary tree with a fixed insertion order is called a complete binary tree.
A history tree is basically a versioned Merkle tree where records are stored left-to-right at the lowest level, in an append-only fashion. History trees are similar to hash chains in the sense that both are persistent ADSs. However, unlike hash chains, history trees also support efficient membership proofs.
Every time an entry is added to a History Tree, a Merkle tree is generated, and a new root hash is computed. The root hash represents a snapshot of the entire log at that point in the sequence. Due to this insertion order, a history tree always grows from left to right. This insertion order enables the reconstruction of old versions of the tree by pruning specific branches from right to left.
We use an index and a layer to identify nodes in a history tree **(N_i,l)**. The index is used to denote the version of the tree. *Leaf nodes* are identified with level 0, and the version of the tree that was produced when inserting the record associated with that leaf (N_i,0). *Interior nodes,* on the other hand, take their index from the index value of their leftmost child. For instance, a version 5 tree has six entry logs stored at the layer 0 ((0,0) to (5,0)) and a depth of 3.

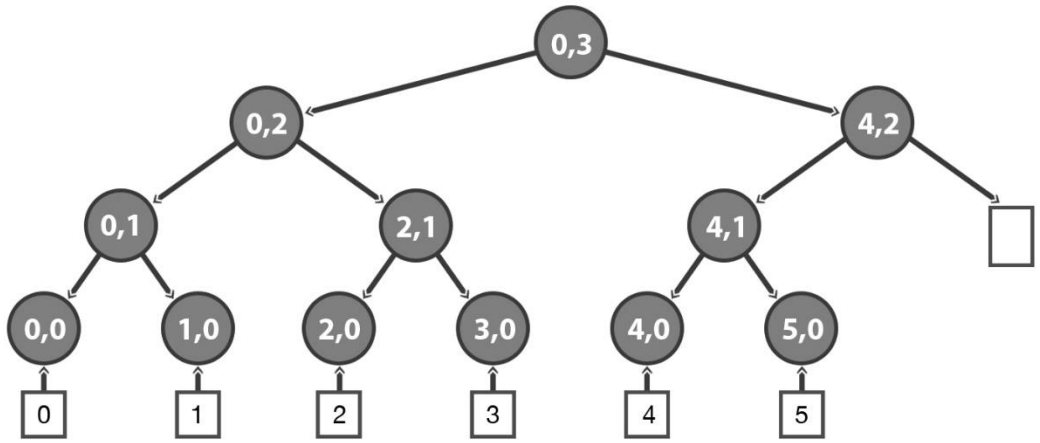

Figure *5* History Tree of Depth 3

Both history trees and ordinary Merkle trees can produce inclusion proofs that allow verifying if a particular log record is present in the log and is consistent with the current snapshot.
Although inclusion proofs provide guarantees about the authenticity and integrity of the stored data, they do not provide any information on the history of a Merkle tree (i.e., how it chronologically evolved). Hence, it is still possible for a malicious server to publish a root hash from a completely different (but technically valid) tree that contains or omits a value. This problem is addressed by

making the tree append-only, hence prohibiting deletion. The append-only nature of the tree then allows for the generation of a *consistency proof* that verifies that a tree root is indeed an extension of an earlier root that is trusted.
Consistency proofs exploit the fact that the perfect subtrees (meaning that all leaves have the same depth) of an append-only binary tree do not change – since a new value is always added as the new rightmost leaf.
A history tree also enables the generation of such proofs, also called as *incremental proofs*. Incremental proofs show consistency between commitments for different versions of the tree. That is, such proofs confirm that all events in the earlier version of a tree are present in the newer one.
A disadvantage of history trees is that they are not searchable. Since a history tree stores event digests at the leaves, it does not allow us to search for such digests efficiently by their contents, as they are sorted by their insertion time (*chronologically*) instead of their value (*lexicographically*).

1.4. Sparse Merkle Trees

Merkle Trees (ordinary or history) do not allow for an efficient query of a particular value, as they both require to traverse each leaf until the data is found iteratively. Furthermore, proving the absence of a value (non-membership proofs) requires recomputing the whole tree; hence all the leaves and their corresponding inclusion proofs must be transferred to the auditor(s).
A Sparse Merkle Tree (SMT) is basically a perfect Merkle Tree of size $2^{|H|}$, where |H| denotes the fixed length of the outputs a hash function H yields. An SMT can either contain an *empty leaf* value, or a leaf associated with a *key-value pair*. The hash value of the key allows searching for the path from the root to the leaf that contains the hashed key as its index. Unlike append-only Merkle trees, SMTs allows for the addition and deletion of nodes, by simply re-labeling the corresponding leaf as empty, or as the added value.
Finally, SMTs can prove the presence of a key-value pair in the tree by generating an inclusion proof of size |H| for the associated leaf with the actual value. Furthermore, SMTs make use of the leaves labeled as empty to prove absence (non-membership proofs).

If we take H to be SHA-256, the resulting Sparse Merkle Tree (a perfect Merkle Tree) would have 2^{256} leaves, therefore clearly not computable. However, in a Sparse Merkle Tree (or SMT), almost all leaves are empty (hence the name sparse). This implies that the parents of these empty leaves are labeled by $H(L_{empty} || L_{empty})$, their parent nodes by $H(H(L_{empty}||L_{empty}) || H(L_{empty} ||L_{empty}))$, and so on. Hence, these digests that only derive from empty leaves are identical for most nodes and can be cached. This means that it is sufficient to compute digests (hash values) individually only for non-empty leaves and their ancestors, making it practical to compute and store.
Compared to append-only Merkle Trees, a Sparse Merkle Tree has a major disadvantage that it cannot create consistency proofs, as SMTs allow for the modification of the tree.

1.5. Verifiable Logs

The first structure described in the *Verifiable Data Structures* white paper [11] published by Google is an append-only *Verifiable Log*. The append-only log property of the log is technically achieved using Merkle Hash Trees. In the beginning, this data structure is empty and is mutated only by entries being appended to it. The append-only property implies that an entry that has been accepted by the log can never be removed or changed afterward. The log periodically publishes a signed tree head, which includes a root hash of all entries for a given log size. Clients of the log can efficiently query the log for proof-of-inclusion of a specific entry. Furthermore, clients can efficiently detect split-view attacks, verify the append-only property of the log, and enumerate all entries held in the log.
The details of the verifiable log data structure are described in RFC6962.

1.6. Verifiable Maps

A verifiable map is a key-value map from a set of keys to a corresponding set of values. This data structure is equivalent to a Sparse Merkle Tree, further explored in Revocation Transparency. The leaf nodes of the tree store the value of a key indexed by an entry index. The index for a key is calculated by taking the SHA256 hash of the key. The tree is, therefore, of the size 2^{256}.

Periodically the map published a signed tree head, which includes a root hash of all $2^{256^{\wedge}2}$ entries. Clients of the map can efficiently retrieve the value associated with a key, verify that this entry is included in the map, and enumerate the whole set of key/value pairs stored in the map.

1.7. Verifiable Log-Backed Maps

As the name implies, a Verifiable Log-Backed map is a Verifiable Map backed by a Verifiable Log. The basic idea is that the verifiable map is populated (in a verifiable manner) by applying the operations contained in the log in order. Moreover, all the commitments of the map (signed tree heads) are published in a separate Verifiable Log to be used for auditing purposes.
This data structure allows for the client to receive verifiable answers and consistency guarantees.
An important feature of this data structure is that the log, the map, and the Signed Map Heads log may be maintained and operated by different parties.

Table 1 compares the verifiable logs, verifiable maps, and verifiable log-backed maps data structures in terms of the operations that they allow and their efficiency.

Table 1 Verifiable Data Structures Comparison as shown in [12]

	Verifiable Log	**Verifiable Map**	**Verifiable Log-Backed Map**
Prove the inclusion of value	Yes*	Yes*	Yes*
Prove non-inclusion of value	Not practical	Yes*	Yes*
Retrieve provable value for a key	Not practical	Yes*	Yes*
Retrieve provable current value for a key	Not practical	No	Yes*
Prove append-only	Yes*	No	Yes*
Enumerate all entries	Yes**	Yes**	Yes**
Prove correct operation	Yes*	No	Yes**
Enable detection of a split view	Yes*	Yes*	Yes*

*This operation can be done efficiently.
**This operation requires a full audit.

2. Current Initiatives

In practice, there are multiple alternatives to achieve a similar functionality of a transparent log. These alternatives range from straightforward technologies, such as storing signed data into a database to far more complicated approaches like blockchain-based technologies.

2.1. Certificate Transparency

Certificate Transparency publishes TLS certificates in a transparent log. Clients such as Google Chrome make use of efficient membership queries to verify that a certificate is logged in a known log before accepting the certificate. Furthermore, auditors can scan the log at any time to detect misissued certificates.
Certificate transparency makes use of chronological append-only log as an authenticated data structure. This log can efficiently create cryptographic proofs regarding the inclusion of a certificate in the logs. Clients of the log can, therefore, verify that a certificate is included in the log (inclusion proofs) and that the current log is an extension of previous logs (consistency proofs). However, CT

does not allow to query the most recent certificate of a particular domain or to prove that a certificate has been revoked (non-membership proofs).

2.2. Key Transparency

Key transparency is basically a large-scale database of accounts (addresses) and their associated public keys. It is mainly comprised of two components: a lookup service for generic records and a public, tamper-resistant audit log of all record changes. Even though it is publicly auditable, only specific IDs, i.e., the key owners, can get a response to their queries and reveal individual records.

The architecture makes use of a Verifiable Log combined with a Verifiable Map, as described in Section 3. The main use case for Key Transparency is *a public key discovery service* to authenticate users. This is extremely important for products that offer end-to-end encryption, as they need a secure way to secure the public keys of recipients. By using this discovery service, account owners are enabled to reliably see what keys have been associated with their account. Furthermore, senders can also use this service to see how long an account has been active and stable before trusting it [3].

2.3. Revocation Transparency

Revocation Transparency (RT) is similar to Certificate Transparency (CT) but serves the purpose of storing certificate revocation information. What mainly distinguishes these two initiatives is that RT requires a ***recent*** status of the certificate. Two plausible options to achieve this are: either the server retrieves this information regularly and serves it to the client along with the certificate, or the server sends a revocation list to the clients.

Also different from CT, RT allows for ***deletion*** and ***update*** of records. What RT requires in its core is an efficient update mechanism that allows the monitors to focus on the updates and not have to download the whole recordset and process it frequently. It still provides efficient membership queries that enable the client to verify whether an entry is contained in one particular ***version*** of the log.
The main purpose of RT is to efficiently ensure that the list of revoked certificates a client sees is the same as the list everyone else sees and that every revocation data is publicly auditable. For RT there are also two possible mechanisms, a sparse Merkle Tree-based mechanism and a traditional Merkle tree-based one that stores pairs from a sorted list of all revoked certificates in its leaf nodes.

2.4. QED

QED is open-source software that was developed by BBVA Labs that implements a transparent log and allows for the establishment of trust relationships by leveraging verifiable cryptographic proofs. QED implements its transparent log by a forward-secure append-only persistent authenticated data structure. Similar to the other initiatives mentioned in this section, every append operation results in the creation of a cryptographic structure (a signed snapshot), i.e., a proof for the append operation, which can also later be used to

i. Verify whether a record is included in the QED Log or not,
ii. Verify that the appended data is consistent, in insertion order, to another log entry.

QED can be requested to prove whether the above statements are true or false for a specific piece of data. In response to that request, QED returns a cryptographic proof, which, combined with the original piece of data, can generate the cryptographic value of the original snapshot [14].

3. Use Cases

The main intuitive use case is the usage of these data structures for registering financial or non-financial data of an organization. The potential use cases, however, are not limited to this data and can be extended to any sensitive events that happen in an organization, or between peers. Examples

include Data transfers, System/Application/Business Logging, Distributed Transactions, etc. In this section, we list some concrete use cases that prove these data structure constructions beneficial.

Verifiable Logs Cases
Verifiable Logs can benefit any organization that issues some records. Examples of such records may include digital certificates (as shown in the CT initiative), binary software artefacts, or even university degrees or access records, insurance policies, or parking tickets. Transparent logs of such records would significantly decrease the number of improper issuances, and discourage insider fraud, as every issuance will be logged in append-only, tamper-proof systems.

Verifiable Maps Use Cases
Verifiable maps can basically benefit every organization that maintains any registry that maps keys to values, where the values are subject to change over time. Examples include:

- a bank maintaining a balance amount for each account number,
- a land registry maintaining ownership information for a specific parcel of property,
- a Certificate Authority maintaining the state as to whether a certificate has been revoked or not.
- An organization maintaining a list of which public keys are in effect per user.

4. References

[1] [Online]. Available: https://www.certificate-transparency.org/.

[2] E. K. Ben Laurie, „Revocation Transparency,“ [Online]. Available: https://www.links.org/files/RevocationTransparency.pdf.

[3] [Online]. Available: https://github.com/google/keytransparency.

[4] [Online]. Available: https://www.certificate-transparency.org/.

[5] [Online]. Available: https://www.rijksoverheid.nl/ministeries/ministerie-van-binnenlandse-zaken-en-koninkrijksrelaties/documenten/rapporten/2011/09/05/diginotar-public-report-version-1.

[6] R. Tamassia, „Authenticated Data Structures,“ in *European Symposium on Algorithms*, 2003.

[7] [Online]. Available: https://info.phishlabs.com/blog/49-percent-of-phishing-sites-now-use-https.

[8] [Online]. Available: https://www.certificate-transparency.org/how-ct-works.

[9] [Online]. Available: https://en.wikipedia.org/wiki/Hash_chain.

[10] [Online]. Available: https://medium.com/ontologynetwork/everything-you-need-to-know-about-merkle-trees-82b47da0634a.

[11] B. L. A. C. Adam Eijdenberg. [Online]. Available: https://github.com/google/trillian/blob/master/docs/papers/VerifiableDataStructures.pdf.

[12] [Online]. Available: https://unicode.org/cldr/utility/confusables.jsp.

[13] [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Website+Phishing.