A-SIT Technology
E-Government Innovation Center

28 September 2020 Study · Privacy & Crypto · Transparency · 14 min read

Privacy-preserving Verifiable Data Audits

Archived. This content is no longer updated. It remains available for reference.

Source https://technology2.a-sit.at/en/publications/privacy-preserving-verifiable-data-audits/

The main aim of this project was to build a prototype system that adds transparency to the way different parties interact with sensitive data by maintaining Verifiable Privacy Audit Logs. The system records a log containing timestamped information of the participants involved in the privacy-sensitive data processing event in a verifiable (append-only, tamper-proof) data structure every time such an interaction takes place. Verifiable Data Structures recently exploited for projects such as Certificate, Key or General Transparency were investigated for the purpose of building such a system.

1. Introduction

As data analytics technologies proliferate, service providers regularly capture, store and analyze customer data for a range of purposes, including learning about their customer's behavior, predictive analysis, and making more informed business decisions. Furthermore, in many situations, the organizations that generate or store the data outsource such data services from external data processors. In such environments, improper data sharing, or modification of shared data, might result in harmful consequences such as breaches of confidential data, or incorrect decisions based on tampered data. Hence, there is a growing need for efficient means to hold data processors accountable for their actions. Such technological means would enable data subjects, data processors/controllers, and third-party auditors to either provide evidence, or verify that the sharing and processing of personal data has happened in a compliance with the data subject's consent, the privacy policies, and the respective data protection legislations. The goal of this project is to implement one of the main building blocks for such a *Verifiable Data Audit* system, namely the storage layer. This system provides visibility on private data access, disclosure and processing events, such that every time such an interaction takes places, it adds timestamped information about the event to a verifiable, append-only, tamper-proof Log. In order to do so, we investigated Verifiable Data Structures recently exploited for projects such as Certificate [1], Key [2] or General Transparency [3]. We further make use of the core transparent data store functionalities provided by Trillian [4], which is an open-source implementation of the Verifiable Data Structures described in [5]. The following sections provide details of the relevant data structures employed in this project, the main functionalities provided by Trillian. We further go into the details of the prototype implementation and finally provide some performance results recorded when evaluating the developed application.

2. General Transparency

Google's approach to PKI, Certificate Transparency (CT), aims to add transparency to the certificate issuing process. CT attempts to remedy certificate-related threats by adding every newly issued certificate to a public log of certificates. This log is append-only and cryptographically secure, which means that certificates cannot be arbitrarily inserted at any point, nor can they be deleted or changed afterwards. Furthermore, the log is publicly verifiable. The log can provide a cryptographically secure proof that a particular certificate has been legitimately appended to the log. What enables CT to provide such strong guarantees is the backing Merkle Tree implementation for the append-only log.

The ideas that support CT, and also other initiatives such as Revocation or Key Transparency, are not specific to their application domain, and can be used to add transparency to almost any trust model. For this project, we applied the same concepts to enable transparent private data access logging. In this setting, data subjects use the services provided by service providers and disclose personal data to the services (a.k.a. data controllers). In turn, data controllers record every personal data processing transaction in a dedicated transparency log server. Data subjects, or even thirdparty auditors, can further retrieve relevant log entries from these log servers, and verify that this data has not been tampered with. Figure 1 gives an overview of how transparency logs can enable the verification of data processing and sharing events in a distributed setting with several users (data subjects) and service providers (data controllers).

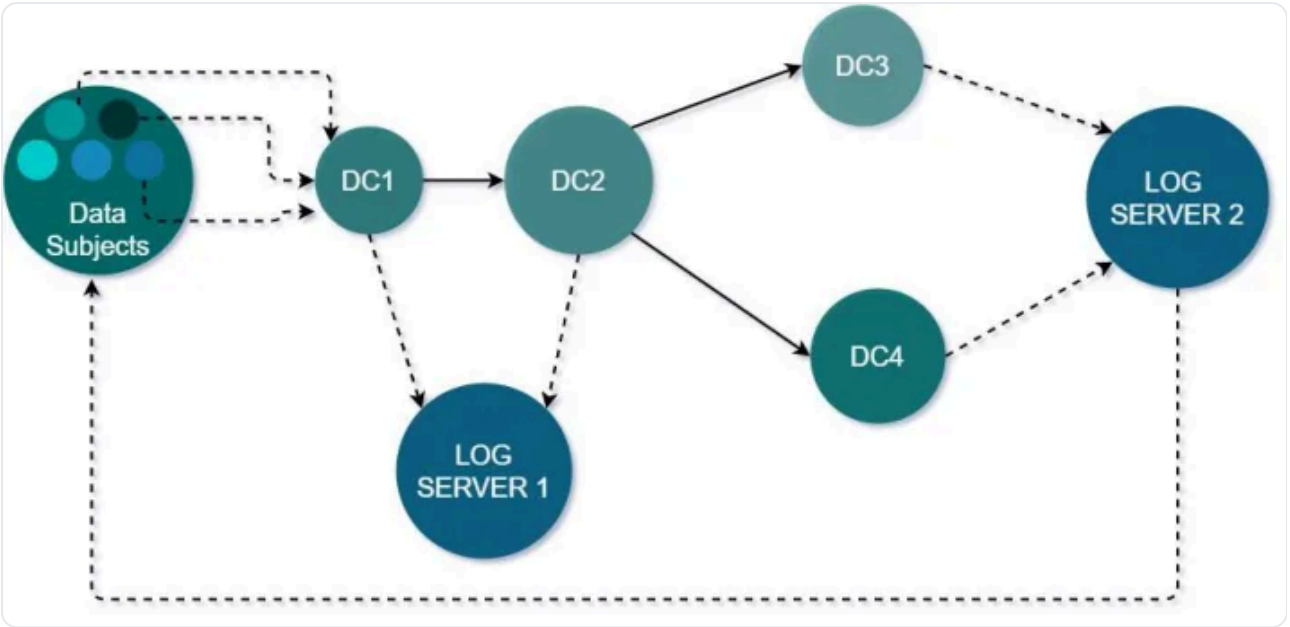


Figure 1: Data Transparency Logging Setting For this project, we built our prototype based on Google's Merkle tree implementation, Trillian. The upcoming section goes into the details of the design and functionalities provided by Trillian, as well as the underlying verifiable data structures that it implements.

2.1. Trillian

Trillian implements the data structures detailed in [5] to allow for transparent, append-only logging of arbitrary data. As such, it implements a Merkle Tree store (and primitives to interact with it) which operates in mainly two modes: an append-only log mode, and a map mode. A third combination of the first two modes is also possible, also called as log-backed map mode. Trillian is implemented as a gRPC service that receives requests over gRPC to either insert or retrieve the respective Merkle Tree data from a separate

storage layer. In order to be able to use the core services provided by Trillian, every application needs to provide additional code (a.k.a. a personality) for application-specific functionality, such as admission criteria, data canonicalization, and external API. On top of the Merkle Tree implementation, Trillian provides two main modes of operation, an append-only Log mode, and a key-value Map mode.

2.1.1. Log Mode The Merkle Tree operating in the Log mode provides an append-only collection of items. It can further operate in two sub-modes: normal (where Trillian assigns sequence numbers to incoming data) or pre-ordered (the sequence numbers are pre-defined by the application). The underlying verifiable data structure is a Verifiable Log, a binary Merkle Hash tree that is filled from the left, resulting in a dense Merkle Tree. The Merkle tree takes as input a list of data entries. Each of these entries is then hashed with a hash algorithm to form the leaves of the tree. Parent nodes are labeled by computing a hash function on their children's hashes concatenated left to right. Figure 2 illustrates an example of a Verifiable Log data structure.

Figure 2 Verifiable Log

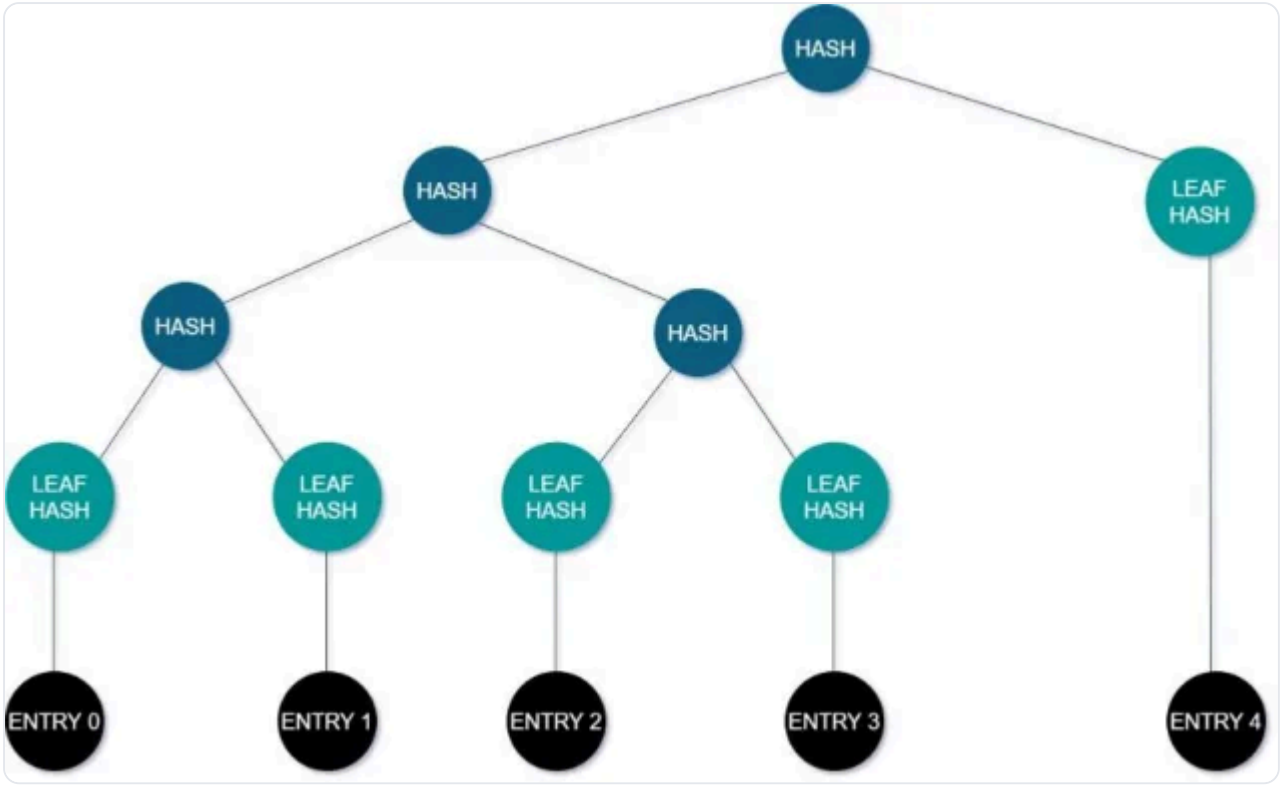


Figure 3: Trillian Log Design, originally produced in [4] 2.1.2. Map Mode A Merkle Tree operating in Map mode, serves the purpose of mapping a set of keys to the corresponding set of values. More specifically, it maps the hash value of the key to the corresponding value. Such a Merkle tree can then either have an empty leaf value, or a full one associated with a key-value pair. Since the hash value of the key is used to define the index of the tree entry, the tree will have the size of 2 [|H|], |H| denoting the length of the output of the hash algorithm. Even though a tree of this size would be uncomputable, most of the nodes of the tree are empty, resulting in a sparse Merkle Tree with an efficiently computable root hash.

The log-based system consists of three main components: (i) a client component (a.k.a. the Personality) that defines the externally visible API of the application, translates requests and responses to gRPC calls, (ii) a server component (Trillian Server) which handles these requests such as queueing a leaf to be added to a tree, or returning consistency and/or inclusion proofs, and (iii) a signer component (Trillian Signer Server),

which regularly polls the storage layer for new leaves to add, and adds them to the Storage layer according to a predefined batch size, in the order that they were queued by the server component. The Signer component is also responsible for calculating and publishing the new Signed Tree Head (STH), which serves as proof that the relevant entries were actually added to the Log. The Server and the Signer component only interact with each other through the storage layer. Trillian minimizes the storage requirements by storing many subtrees for a given depth. The reason behind this is that the hashes of perfect subtrees do not need to be recomputed and can be stored long-term. Hence, Trillian stores only the roots of the largest perfect subtrees, and consequently uses these stored hashes to compute the root of the overall tree. Figure 3 provides an overview of the design of a Trillian implementation of the Verifiable Log.

It is the hash value of the key that enables these trees to be efficiently queried for a particular value. The same is not true for append-only logs, as they require to traverse each leaf until the data is found iteratively. Furthermore, maps also allow for efficient proof of the absence of a value (a.k.a. nonmembership proofs) by making use of the leaves that are labelled as empty. On the other hand, an append-only log requires recomputing the whole tree in order to generate such a non-membership proof, as all the leaves and their corresponding inclusion proofs must be transferred to the auditor(s). Compared to append-only logs, maps cannot generate consistency proofs, as they allow for deletion and modification of a leaf node, by re-labelling it as empty, or as the newly added value. Figure 4 gives an example of a Verifiable Map.

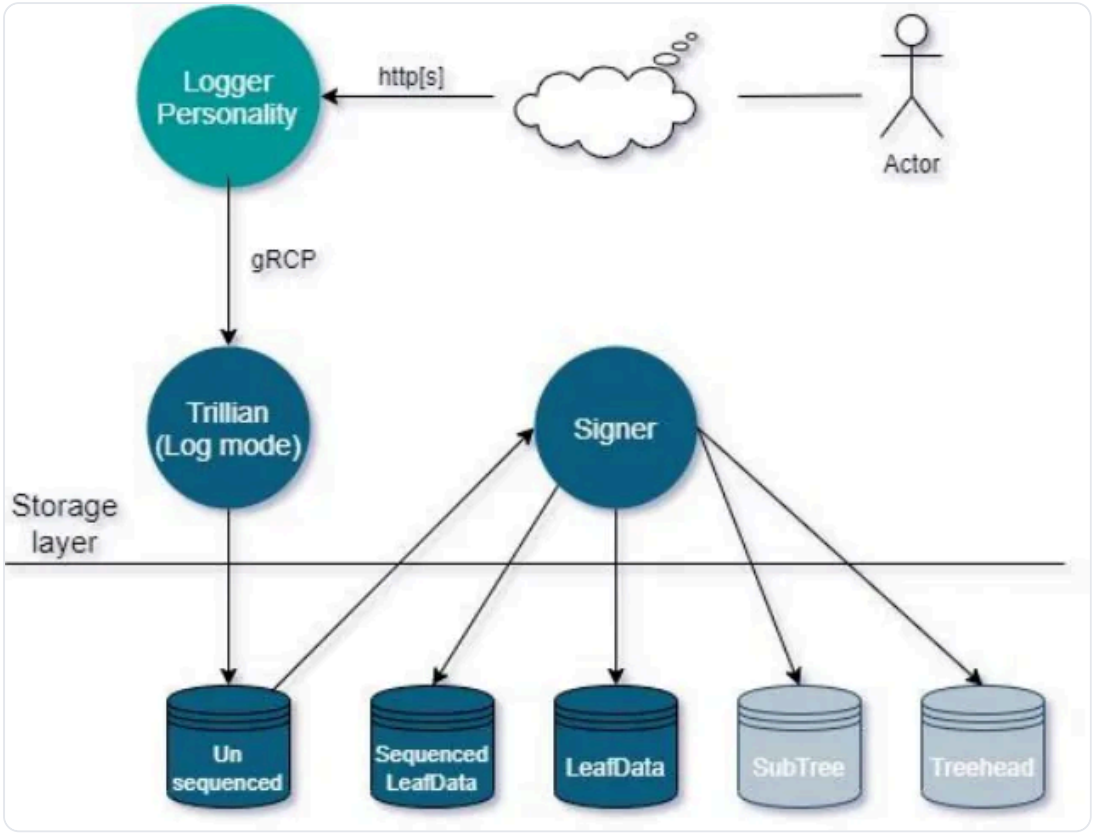


Figure 4: Verifiable Map (Sparse Merkle Tree) In Trillian, maps are operated similar to the append-only Logs, with the difference that there is a single Server component (Trillian in Map mode) that interacts with the Map Personality to handle user requests, and consequently with the storage layer to further insert or retrieve Map values, or requested proofs, accordingly.

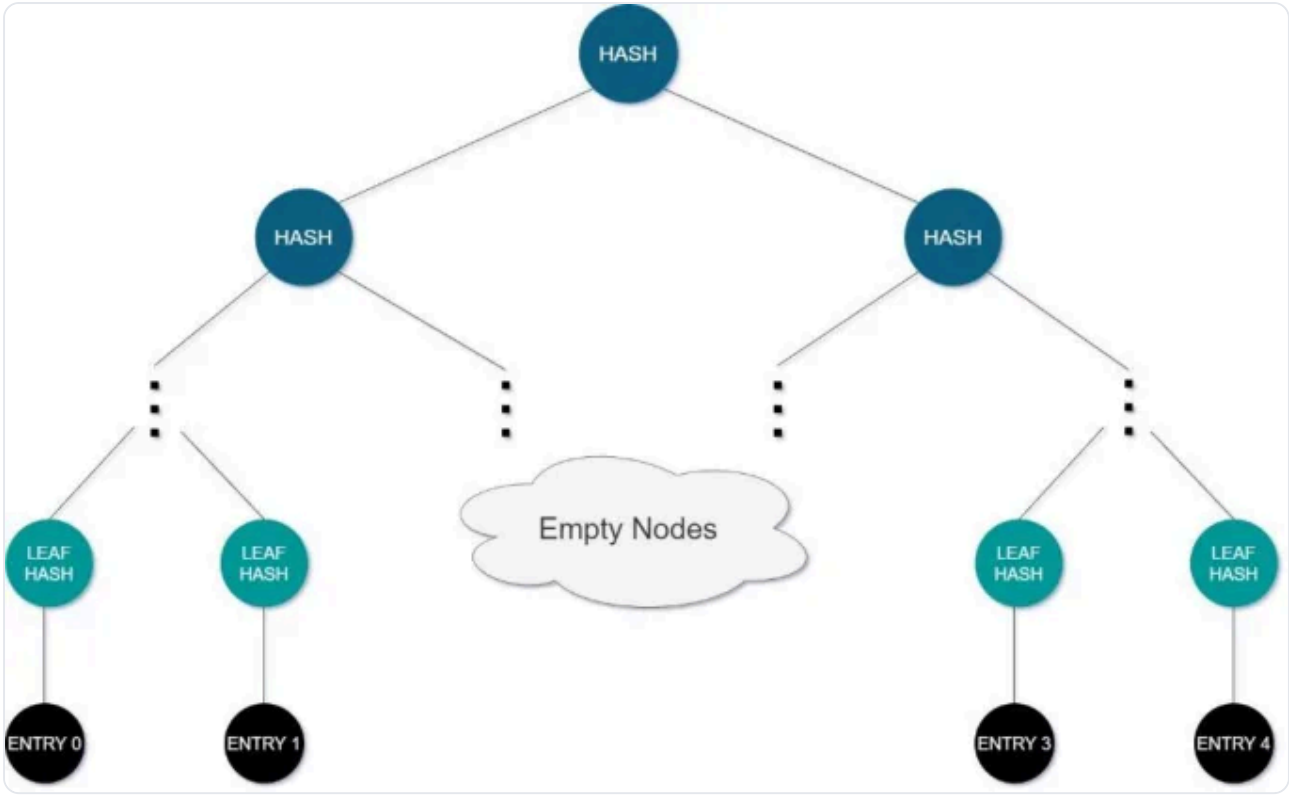


Figure 5: Trillian Map Design, originally produced in [4] 2.1.3. Log Backed Map Mode Given that Map values can be tampered with, a Map instance cannot be reliably audited or monitored as a standalone component. As a solution to this, a Log and a Map can be combined to for a Log-Backed Map, which is mainly composed of three Merkle Trees:

(i) An append-only log containing all the entries generated by the application, (ii) A verifiable Map, that is populated by entries contained in the Log, according to the order they exist in the Log (iii) A second append-only Log that contains Signed Map Tree Heads as its entries. This would allow for the consistency guarantees that the Map cannot provide as a standalone component.

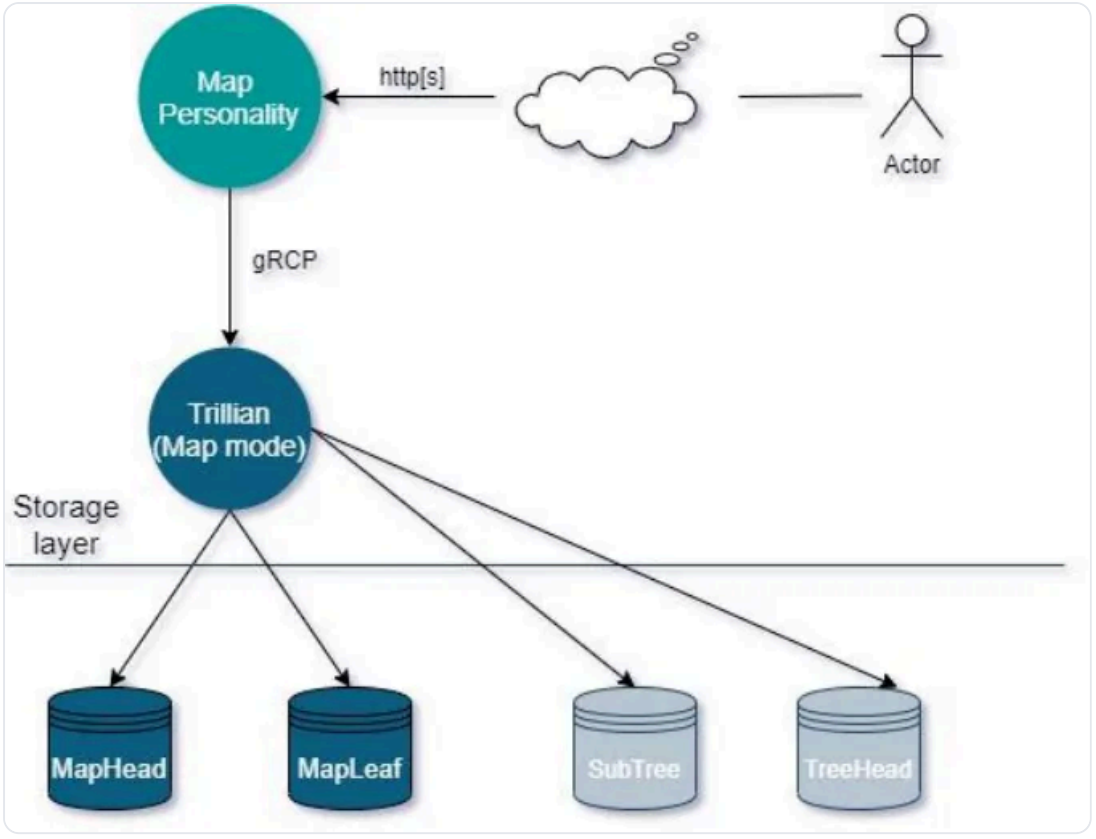


Figure 6: gives an overview of the Verifiable Log Backed Map Data Structure.

(a) (b) (c) Figure 6 Verifiable Log Backed Map composed of (a) a Verifiable Log, (b) a Verifiable Map, (c) a Log of Signed Map Heads Figure 7 illustrates the design of such a Verifiable Log Backed Map in Trillian. A second Trillian personality, i.e. application specific code, is required to map the Source Data from the Log to the Map. Finally, the Map will publish each root hash of a new revision to the second Trillian Log.

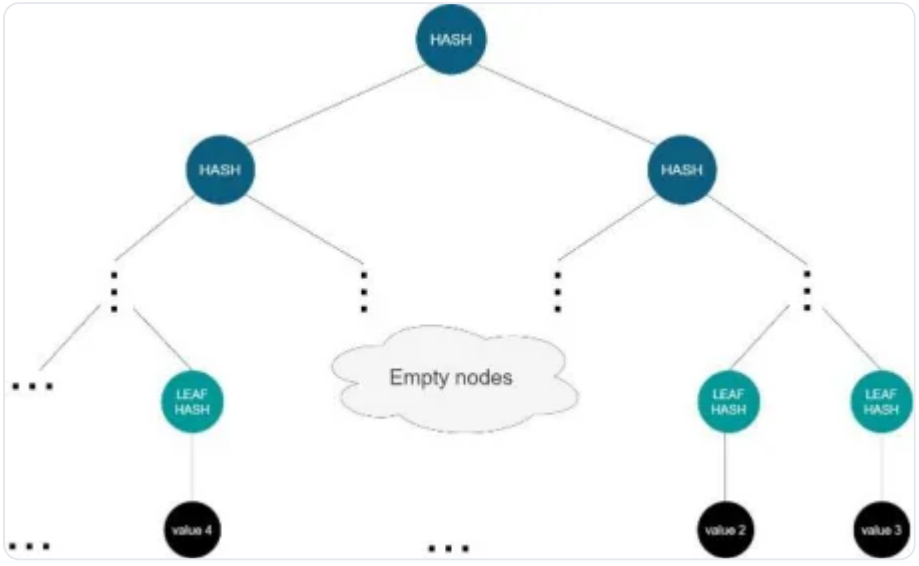
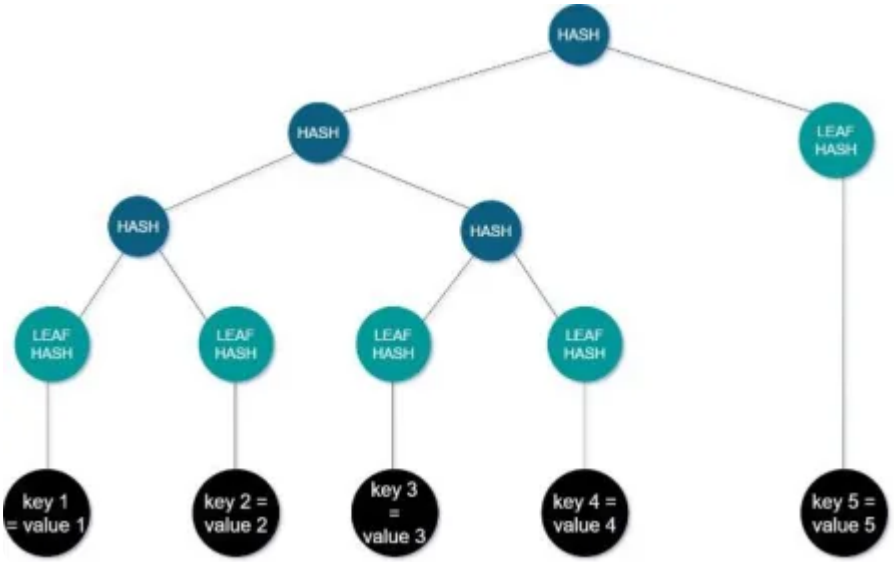
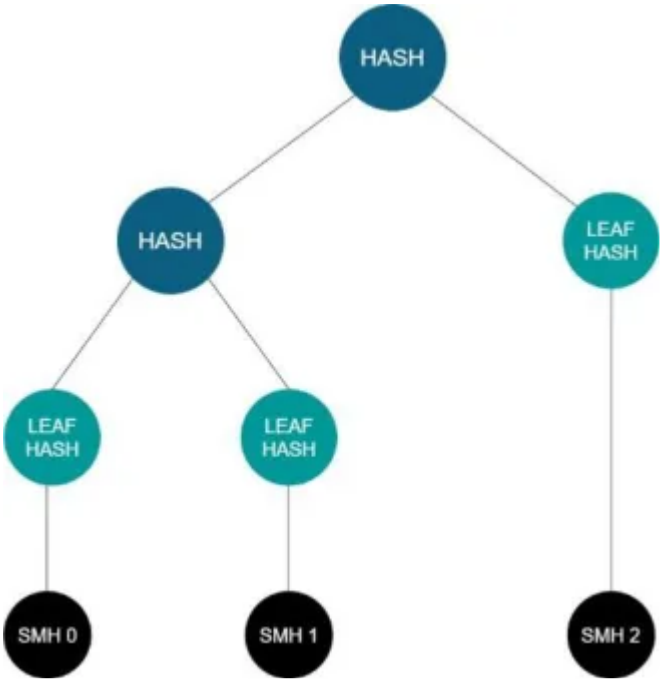
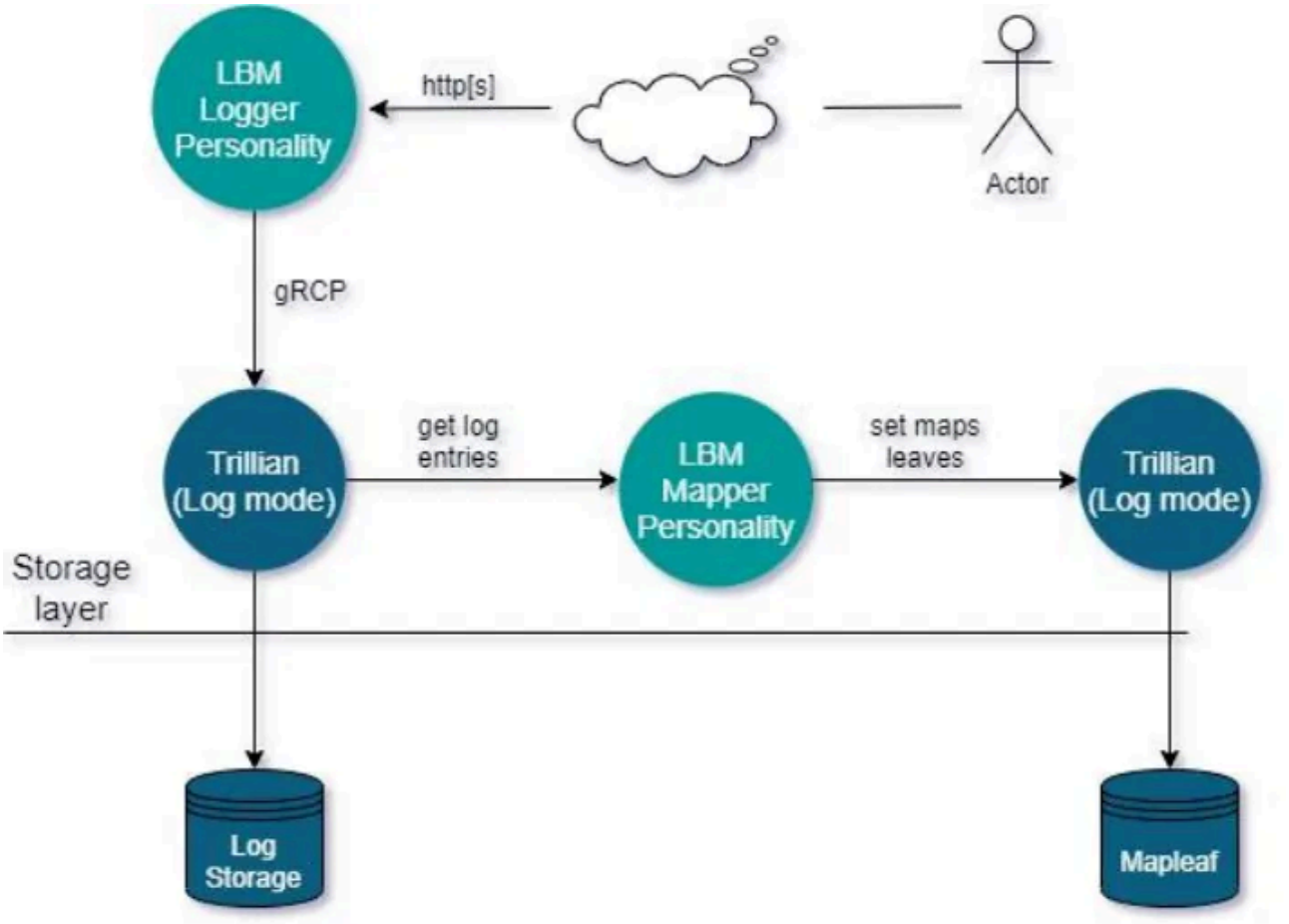


Figure 7: Log Backed Map Design, originally produced in [4]


HASH
HASH
LEAF HASH
HASH
HASH
LEAF HASH
LEAF HASH
LEAF HASH
LEAF HASH
key 1 = value 1
key 2 = value 2
key 3 = value 3
key 4 = value 4
key 5 = value 5


HASH
HASH
LEAF HASH
LEAF HASH
LEAF HASH
SMH 0
SMH 1
SMH 2



3. Implementation

For this project, a transparency demo application was built using the core transparent data store functionalities provided by Trillian. We developed a reliable, verifiable transparency service that consists of a Logger Personality that gets sample data to be logged from a publicly available register [1], adds the data to the Trillian append-only Log, and further maps the data from the Log to a Map, by implementing a mapper personality. Finally, the data extracted from the map is served through a webserver.

3.1. Initial Setup

The application was developed using Go. MySQL was used to provide the data storage layer of the application. The sample data was encoded using the JSON encoding format. Our personality interacts with Trillian via gRPC calls (i.e. the personality serves the purpose of translating HTTPS+JSON calls to gRPC calls, and vice-versa). Our implementation consists of the following components:

(i) An application-specific Transparency Logger Personality, that handles the user requests to enter/retrieve data to/from Trillian. (ii) A Trillian Server in Log mode, which receives the requests from the application-specific Personality and interacts with the data storage layer to add/retrieve the corresponding entries, or proofs. (iii) A Trillian Signer Server component, that continuously polls the storage layer for pending entries to be added to the actual append-only Merkle Tree, adds them and generates proof of inclusion. (iv) Another application-specific Mapper Personality, that retrieves entries from the Log, transforms and feeds

them to the Trillian Server in Map Mode. (v) A Trillian Server in Map mode, which handles the requests from the Mapper Personality to enter/retrieve key-value pairs to the map Merkle Tree. (vi) A demo webserver, which serves the purpose of presenting data further retrieved from the Map.

3.2. Application Walk-Through

The main steps involved in running our Transparency application are as follows:

i. We first start by running the Log Server and then creating an append-only Log that will hold the data from the register. ii. Our application (personality) connector to the Log Server (using gRPC.Dial()) and then creates a Trillian client. The application then feeds each entry from the register, to the Trillian Server, using the QueueLeaf interface provided by the client and the Log ID of the log that was created in step i. iii. Note that in step ii., the entries are not actually appended to the Log, as nothing gets actually committed without running the Log Signer component. So the Log Signer component is run next. The Signer polls the Storage Layer for pending entries, and adds them to the append-only tree according to a user-defined batch size, and the order they were queued by the Log server component. After the Signer component has processed and added all the pending entries, our application can connect to the Log Servers to further retrieve the entries via the Trillian client interfaces, as well as the corresponding inclusion proofs for them. Note that in this step, it is only possible to retrieve all the entries contained in a log, and not query the log for a particular entry. iv. In order to be able to query our transparent data store, we map the entries already present in the Log to a key-value map. To do so, we use our Trillian client to retrieve the log and iterate through it, one leaf at a time. Our Mapper Personality then maps the entry keys to the entry contents. Afterwards we create a new tree (in Map mode), and start a Map Server to add each entry to the key-value Map.

1. 1 https://www.registers.service.gov.uk/registers/ddat-profession-capability-framework/ v. Finally, our application can connect to the Map Server and is able to query the Map and retrieve values for specific keys. For demonstration purposes, we finally present the retrieved values from a webserver, via the records.json endpoint.

Figure 8 Sample Data retrieved per key from the Verifiable Map

4. Results

This section presents measurements related to the performance of the developed application. We first present micro-benchmarks for the basic operations of our system and then examine the throughput of the system for different batch sizes.

4.1. Micro benchmarks

Table 1 presents micro-benchmarks for the basic operations of our system. Since the system is mainly composed of an append-only Log, and a key-value Map, these operations include state updates (log and map insertions), state retrievals (retrieval of log and map entries), and proof generation (inclusion proof per entry). The measurements were obtained by averaging over the 1000 operations (for all the entries of the sample data register used).

As the results show, entry insertion for the Map-based implementation takes longer when compared to the Log-based implementation. Entry retrieval, however, is much more efficient for the map, given that the entries can be individually queried for using the hash values of the keys. Note that for the Log-based

implementation, the entry retrieval can only be done as part of enumerating all of the entries of the log.

Measures Log Map

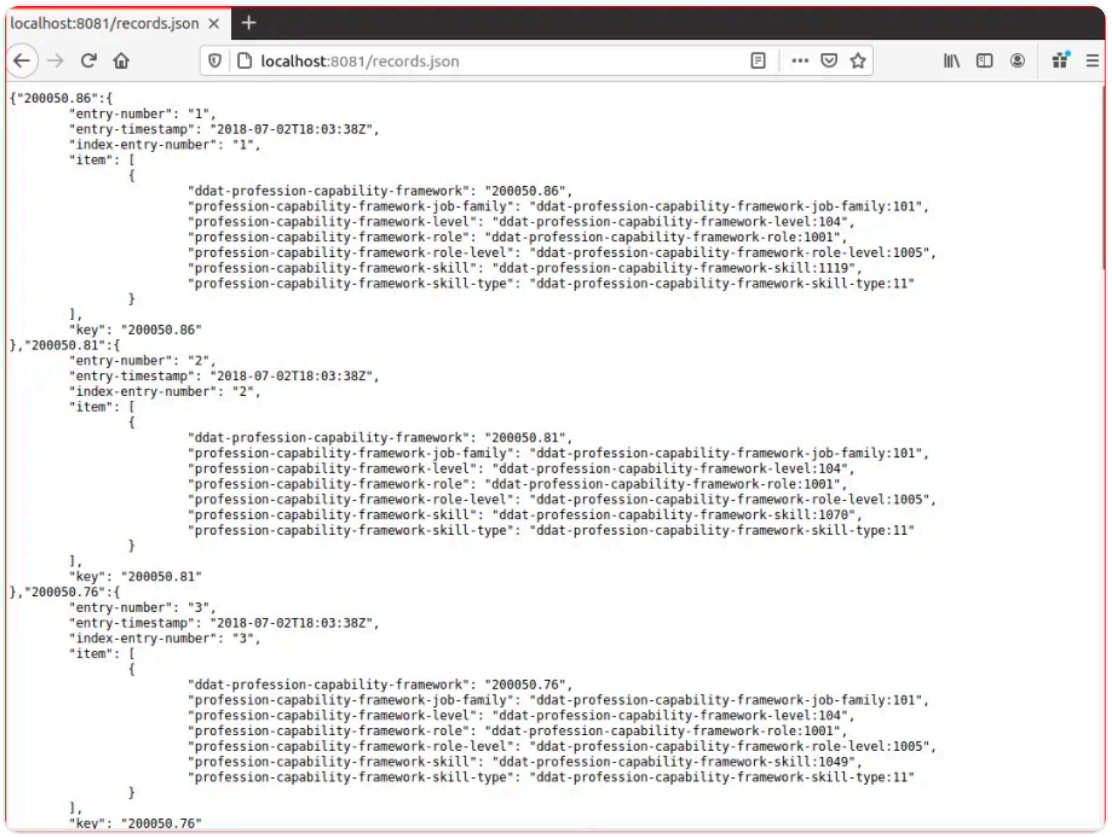


Figure 9: presents a plot of the throughput for different batch sizes of the Log Signer component. The batch size determines how many pending items (queued leaves) at a time the Log Signer server retrieves from the storage layer to insert them to the actual append-only log. There exists a tradeoff between batch size and throughput. A larger batch size will allow for a higher throughput, but a lower batch size might have additional advantages as ensuring the data to appear as soon as possible, which is crucial for some applications.

Table 1 Micro benchmarks of basic operations for the Log-Based and Map-based implementations of the transparent data store

4.2. Throughput

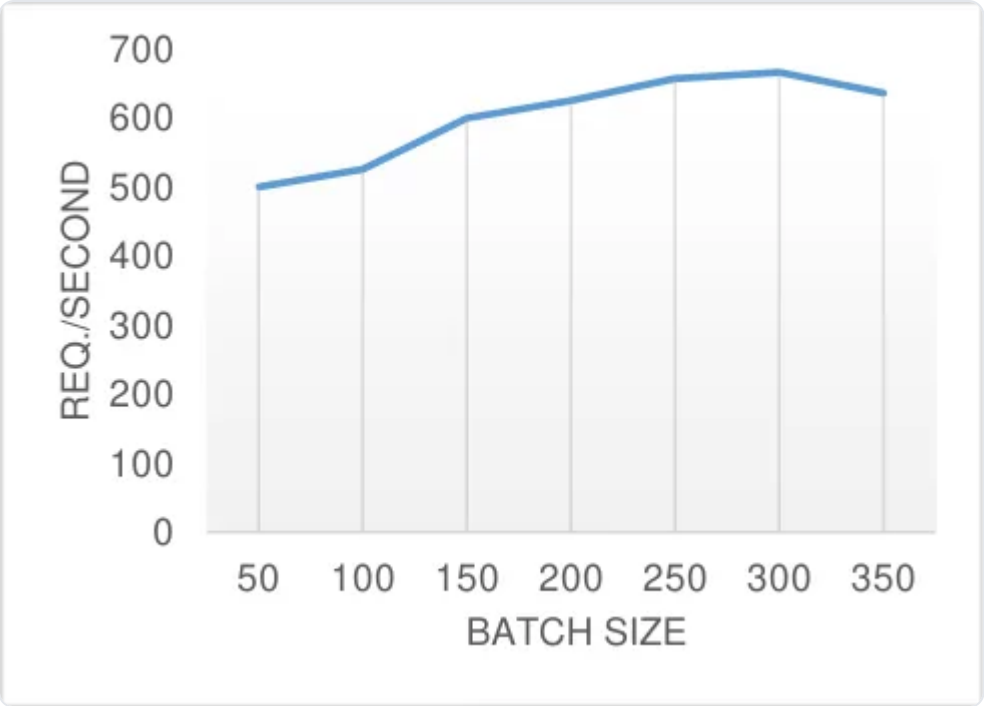
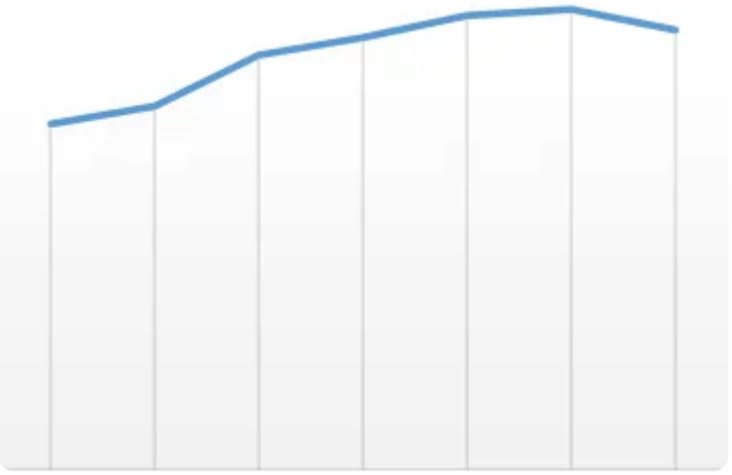


Figure 9: Log Signer Throughput per batch size



5. Conclusions

This project implemented a Verifiable Data Audit application that enables data controllers to securely log private data accesses. For this purpose, we built off Google's Merkle Tree implementation, which provides basic functionalities for a transparent (append-only, tamper-proof, publicly verifiable) data store. Our application provides a verifiable transparency service that gets data to be logged from an external application (data controllers in our use case), adds the retrieved data to the verifiable data structures implemented by Trillian, and finally retrieves the logged data and presents them via a webserver. We recorded micro benchmarks for the basic operations provided by state update, retrieval and proof generation and examined the effect of batch size on the throughput (number of requests processed per time unit) of the system.

6. References

[1] “Certificate Transparency.” https://www.certificate-transparency.org/ (accessed Sep. 18, 2020).

[2] “GitHub - google/keytransparency: A transparent and secure way to look up public keys.” https://github.com/google/keytransparency (accessed Sep. 18, 2020).

[3] “General Transparency - Certificate Transparency.” https://www.certificatetransparency.org/general-transparency (accessed Sep. 18, 2020).

[4] “GitHub - google/trillian: A transparent, highly scalable and cryptographically verifiable data store.” https://github.com/google/trillian (accessed Sep. 18, 2020).

[5] A. Eijdenberg, B. Laurie, and A. Cutter, “Verifiable Data Structures,” 2015.