This paper sets out the cryptographic mechanism for transaction verification within myGaru Platform. myGaru solution design covers the end-to-end personal data lifecycle with a unified transparency system that generates cryptographically signed evidence of data usage at every level, built on well-known, trusted cryptographic constructions and fully auditable by trusted third parties. In practice, the Audit Log records all sensitive data-related activities and provides verifiable proofs of honesty to third parties; operational evidence is accumulated in an auditable manner by a set of software components. To strengthen assurance, the open-source the audit-log engine and essential components are accessible to Partners to enable them to verify that the system behaves exactly as declared.

This document contains:

Chapter 1 — Security model and architecture

High-level overview of where the Audit Log fits within myGaru's architecture and what security guarantees it provides; focuses on append-only logging, cryptographically signed immutable entries, and independent third-party verifiability at both operational and software-design levels.

Chapter 2 — Merkle tree as append-only log cryptosystem

Defines how Merkle trees back append-only logs and which proofs are used for verification.

- **2.1 Merkle tree definitions** — root/leaf/node; authentication (audit) path; RFC 6962 domain separation for leaf/node hashes.
- **2.2 Balanced and unbalanced trees** — growth behaviour and effect on authentication paths.
- **2.3 Permanent and ephemeral hashes** — why siblings should avoid ephemeral nodes; security rationale.
- **2.4 Consistency of trees** — proofs that a later tree is a superset of an earlier one (e.g., N=6 → N=12).
- **2.5 Tiling** — Trillian's storage optimisation using tiles/subtrees, caching and bounded read/write patterns.

Chapter 3 — Cryptographic design: Certificate Transparency background

Explains why the CT model is reused instead of bespoke crypto; outlines roles (client, auditor, monitor), inclusion/consistency proofs, and gossip to detect split-view attacks.

Chapter 4 — Applying Certificate Transparency approach to the myGaru business challenge

Describes how CT principles are adapted to myGaru: transactions (not certificates), stakeholders may act as monitors and auditors, a gossip protocol is required; adds an End-User inclusion check while log creation follows a Trillian-based approach.

Chapter 5 — Implementing Trillian-based audit log

Describes system architecture and verification workflows using Trillian, Elastic, Filebeat, Keystore, and AL Verifier.

- **5.1 Architecture and components** — Filebeat (signs/pushes logs), Elastic (stores), Keystore (keys), Trillian (Merkle tree), AL Verifier (Monitor/Auditor/End-User checks).

- **5.2 Inclusion of transaction info** — inclusion proof vs signed timestamp (MMD) and later retrieval.
- **5.3 End User's check** — inclusion proof for a specific transaction via root + siblings.
- **5.4 Auditor's check** — append-only/consistency verification across successive roots.
- **5.5 Monitor's check** — validation of newly observed transactions against business rules.
- **5.6 Gossip protocol** — exchange of roots, random inclusion proofs, and consistency proofs between AL Verifiers to detect split views.

Chapter 6 — Integrating a Trillian-based audit log

Explains how logging is integrated across services to ensure coverage of sensitive operations.

- **6.1 Current level: always-loggable choke points** — strategic placements where every sensitive operation is inevitably logged.
- **6.2 Future work: business-logic validation** — FSM-based event validation; a separate log to flag anomalies for monitors.

1. Security model and architecture

The following scheme outlines the place of the Audit Log service component in the overall architecture of the myGaru solution. The green color denotes components that interact with the Audit Log service.

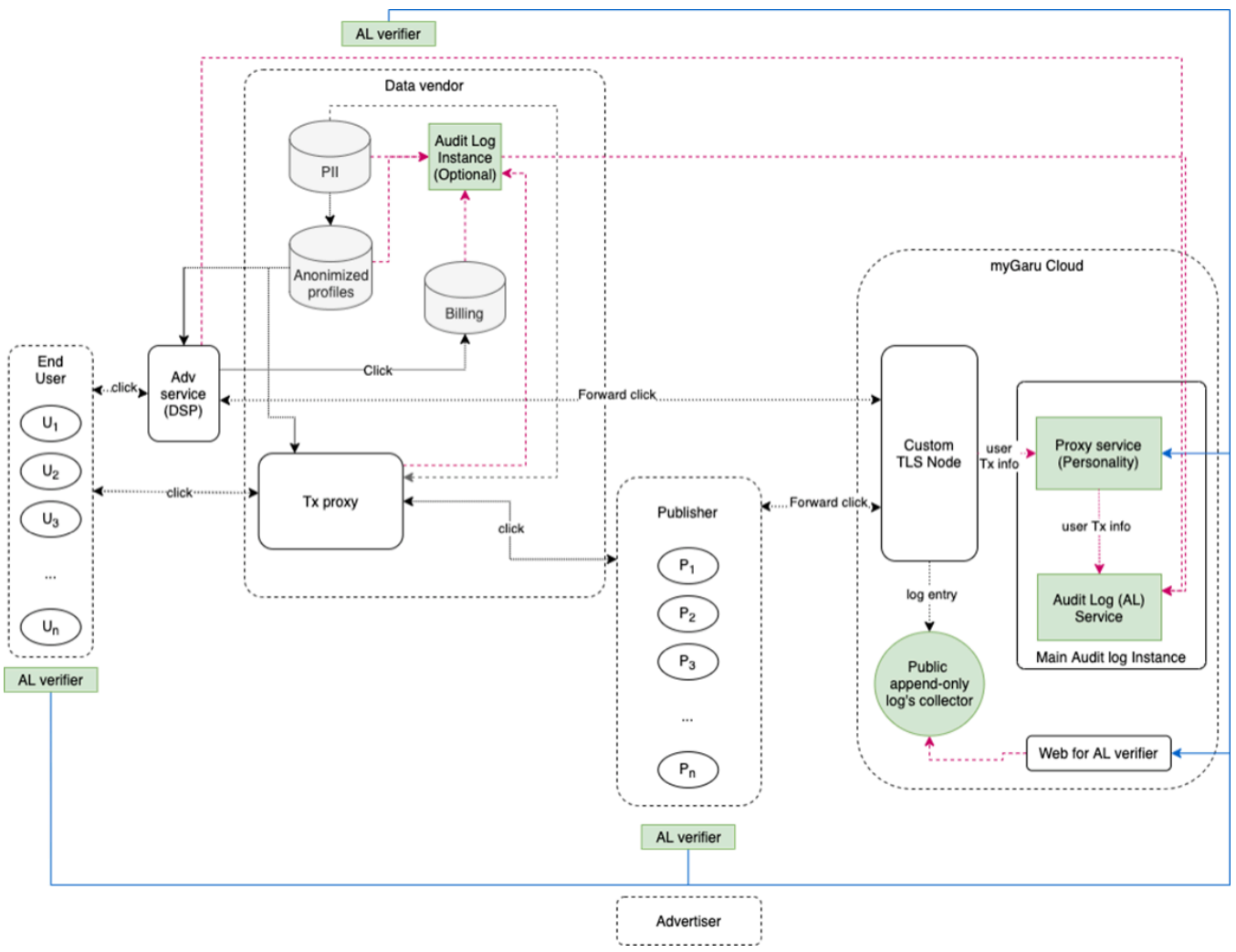

Figure 1. A high-level scheme of the myGaru audit log solution.

Based on incentives and problems outlined above, Audit Log system should provide a number of extended security guarantees:

- append-only log system;
- cryptographically-signed and tamper-proof log messages and their sequence;
- strong cryptographic proofs of integrity/consistency, that can be verified by 3rd parties, independent of core service.

These security guarantees have to be easily verifiable by 3rd parties:

- On an operational level by participation in the Audit Log system as verifiers / monitors and relying on cryptographic proofs.
- On software design level — by evaluating software design and source code to ensure that system's cryptographic claims are backed by sound design and implementation.

2. Merkle tree as append-only log cryptosystem

Warning: this and a few subsequent sections are pretty intense in computer science and basic cryptography. The goal is to provide an easy reference for those who are looking to understand the technology without jumping between different cryptographic books.

2.1. Merkle tree definitions

A Merkle tree is constructed from N records, where N is a power of two. First, each record is hashed independently, producing N hashes. Then pairs of hashes are themselves hashed, producing N/2 new hashes. Then pairs of those hashes are hashed to produce N/4 hashes, and so on until a single hash remains.

This diagram shows the Merkle tree of size N = 16:

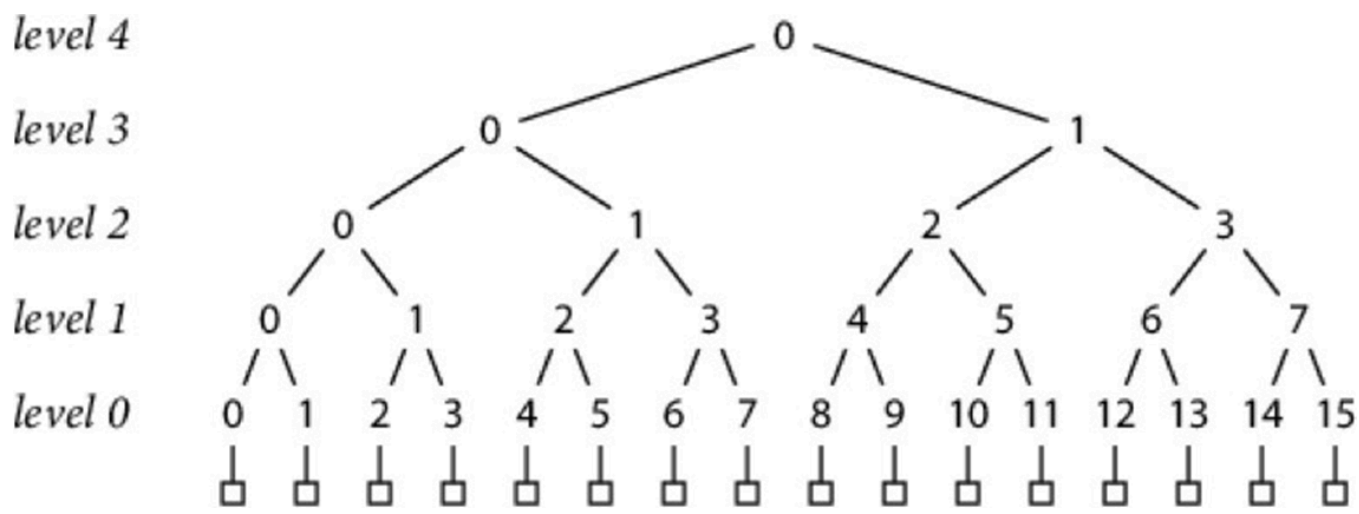

Figure 2. .A Merkle tree that consists of 16 records.

The boxes across the bottom represent the 16 records (log entries in our case). Each number in the tree denotes a single hash, with inputs connected by downward lines.

Root (or anchor) of a tree — is a top-level hash (hash 0 at level 4).

Leaf — is a particular log entry stored in a tree combined with its hash value — level 0 hashes in a diagram above (example above shows 16 leaves).

Depth of the tree above is 4 (log216). The depth increases with the addition of new logs.

In order to verify that a particular log entry (f.e., under hash 9 at level 0) is indeed present in the tree, the following definition is introduced:

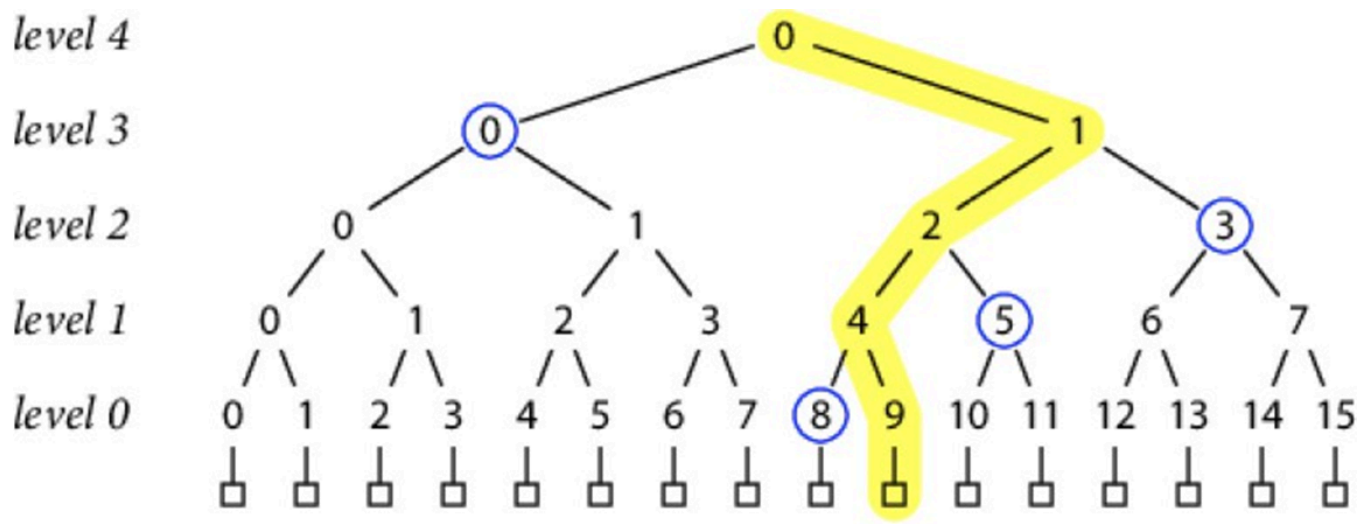

Figure 3. Authentication path of hash 9.

Siblings (or **audit path** or **authentication path**) — a set of hashes that allows computing of the root and comparing it with some value stored in a source of trust.

For the example tree above, siblings include four hashes:

- hash 8 at level 0;
- hash 5 at level 1;
- hash 3 at level 2;
- hash 0 at level 3.

Each hash in siblings allows the recomputing of a parent hash and goes on up to the root. The number of siblings is always equal to the tree **depth**.

Node — is an "intermediate" hash in a tree between lowest-level hashes (leaves) and highest-level (root) in a tree hierarchy — e.g. any hash on level 1, 2 or 3 in the example above, or more specifically:

- hash 0,1,2,3,4,5,6,7 at level 1;
- hash 0,1,2,3 at level 2;
- hash 0,1 at level 3.

Note, that according to RFC 6962, hash-values calculations for leaves and nodes do differ. Hashes for nodes should be computed with 0x01 byte appended to the input, while hashes for leaves should be computed with 0x00 byte appended to the input. This domain separation is required to guarantee second preimage resistance.

2.2. Balanced and unbalanced trees

Data (log entries in our case) may appear in a tree arbitrarily.

The tree is called “**balanced**” if the number of records equals to a power of 2 (2, 4, 8, 16, ...). Otherwise, the tree is called “**unbalanced**”.

The following diagram illustrates balanced and unbalanced trees.

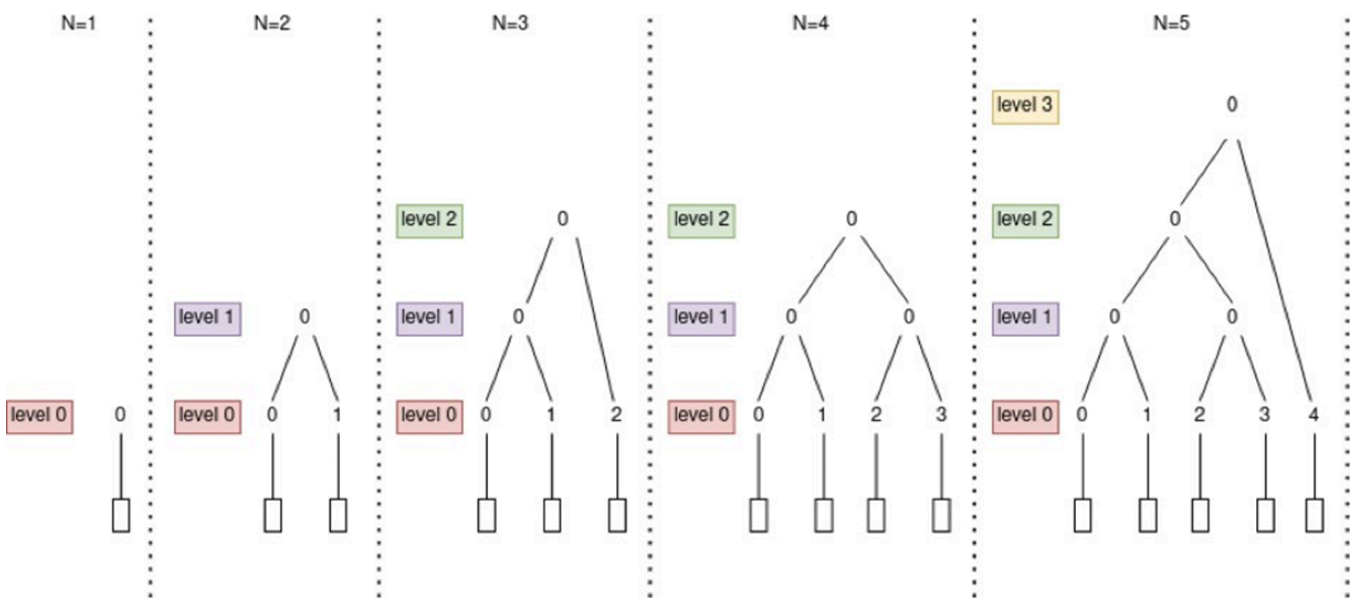

Figure 4. This tree is balanced when N=2 and N=4, and unbalanced when N=1, N=3, N=5..

Let’s take an empty tree. After adding the first log entry (N=1), the tree contains only one element expressed by hash 0 at level 0. Its root is also hash 0 at level 0.

After adding a second log entry (N=2), the tree contains two elements expressed by hash 0 at level 0 and hash 1 at level 0. The root of the tree is recomputed and is turned into hash 0 at level 1. This tree is called balanced since the set of siblings includes exactly one element for both log entries.

After adding a third log entry (N=3), this tree becomes unbalanced. It contains three elements (hash 0, hash 1, hash 2 at level 0). The root of the tree is recomputed again and is turned into hash 0 at level 2. Note, that sets of siblings for hash 0 and hash 1 contain two elements each, while the set of siblings for hash 2 will contain only one element (hash 0 at level 1).

2.3. Permanent and ephemeral hashes

When the Merkle tree grows (by appending new log entries), hash values of its nodes, anchor, and proof of particular leaf inclusion (siblings for particular leaf) are subject to change (in contrast to level 0 hashes that will never change once included).

Let's imagine a tree built on top of 13 elements.

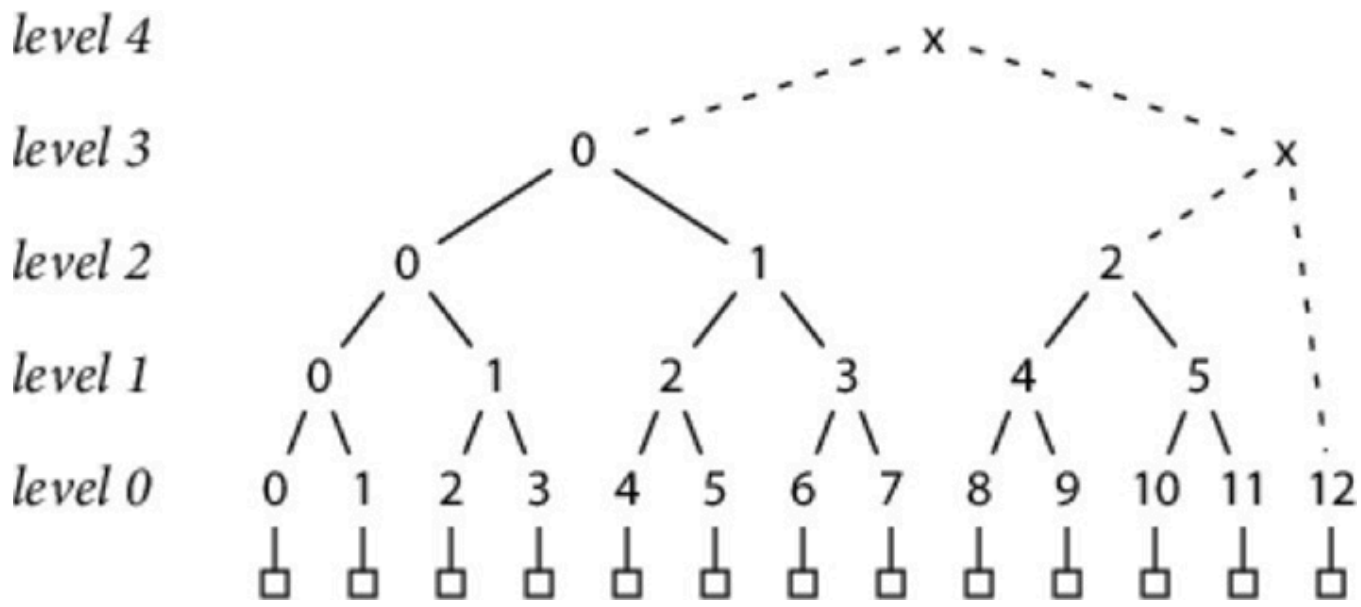

Figure 5. This tree has two ephemeral hashes marked as “x”.

The numbered hashes are **permanent**, in the sense that once such a hash appears in a tree of a given size, that same hash will appear in all trees of larger sizes.

In contrast, the “x” hashes are **ephemeral**; they are computed for a single tree and never seen again.

Avoiding ephemeral nodes in siblings is possible at a small cost of sibling length and essential from a security perspective to guarantee append-only properties of the log.

In order to check the 4-th leaf inclusion, instead of using siblings,

- hash 5 at level 0;
- hash 3 at level 1;
- hash 0 at level 2;
- hash “x” at level 3.

We can use a slightly longer set:

- hash 5 at level 0;
- hash 3 at level 1;
- hash 0 at level 2;
- hash 12 at level 0;
- hash 2 at level 2.

2.4. Consistency of trees

Merkle Tree structure has another property important from security perspective — consistency proof. Consistency proof is a way to check if the later tree (with root T_2) includes everything in the earlier tree (with root T_1), in the same order, and all new elements come after the elements in the older version.

The idea of **consistency proof** is to present a verifiable calculation of T_1 (it should be based on all leaves of the older tree), and a verifiable calculation of T_2. For example, consider the N=12 tree from the previous diagram. We want to check if the N=6 tree is a subtree of it:

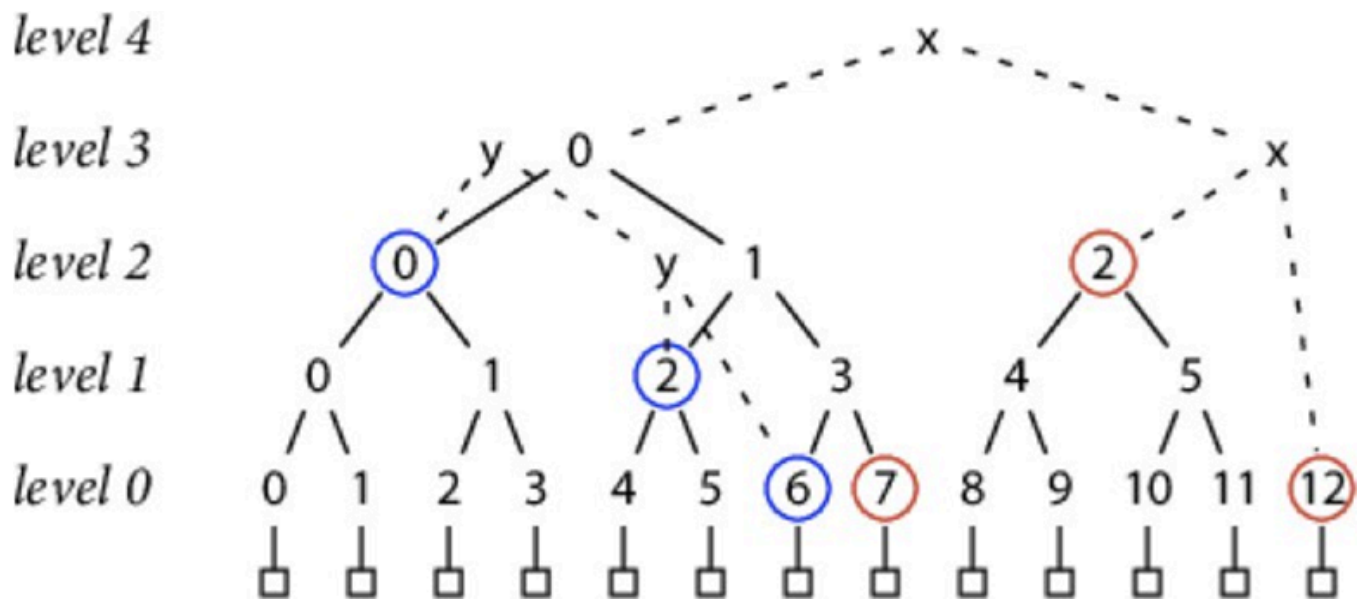

Figure 6. Consistency proof: checking if a N=6 subtree is a part of N=12 tree.

The proof should contain elements (circled in blue), that allow recomputing the ephemeral root of the N=6 tree, which is a hash “y” at level 3. Those elements are following:

- hash 6 at level 0;
- hash 2 at level 1;
- hash 0 at level 2.

Typically, audit log software operates in a distributed infrastructure (multiple audit log components run on multiple servers), meaning that some audit log components have access to some part of the tree (T_{1a}, T_{2a}, etc), and only “source-of-truth” components have access to the whole tree (T_2).

These components are described in a chapter below *5.1 Architecture and components of Trillian-based audit log system*. “Audit log components that perform consistency checks” are called AL Verifiers, “source-of-truth” component that stores the whole tree is called the Trillian server. To learn more about dividing trees into subtrees refer to a *2.5 Tiling.*

Consistency check is performed by an audit log component that has access to some part of the tree and requires to prove that it operates on a consistent tree.

Assuming that the audit log component knows the root of the N=6 tree, it can recompute hash “y” at level 3 using the data from the source-of-truth component and comparing it with its own. This is the first part of the consistency check.

Next, as we mentioned above, the proof should additionally contain elements (circled in red), that allow recomputing the root of N=12 tree (using elements necessary for computing root of N=6 tree, which we already defined), which is hash “x” at level 4.

Those elements (together with the previous elements) are following:

- hash 7 at level 0;
- hash 2 at level 2;
- hash 12 at level 0.

Assuming that the audit log component has computed the root of the N=12 tree, it can recompute hash “y” at level 3 using the source-of-truth-provided data and compare it with its own. This is the second part of the consistency check.

The total proof would contain six elements:

- hash 6 at level 0;
- hash 2 at level 1;
- hash 0 at level 2;
- hash 7 at level 0;
- hash 2 at level 2;
- hash 12 at level 0.

Where the three first elements are used to check the correctness of N=6 tree root and are also a necessary part of the input for N=12 tree root checking. In RFC 6962 Section 2.1.3 one can find additional examples.

2.5. Tiling

Tiling principle is used for storage optimization of the Merkle Trees, suggested by Trillian.

Storage optimization is possible due to a predictable access pattern to the data stored in the Merkle Tree. Moreover, it is known for sure that 1) tree grows only to the right; 2) completely populated left subtree of the tree structure is never further mutated; and 3) inclusion proofs contain only permanent nodes.

The core idea of optimization is storing the tree as a collection of subtrees (or tiles) instead of storing raw nodes.

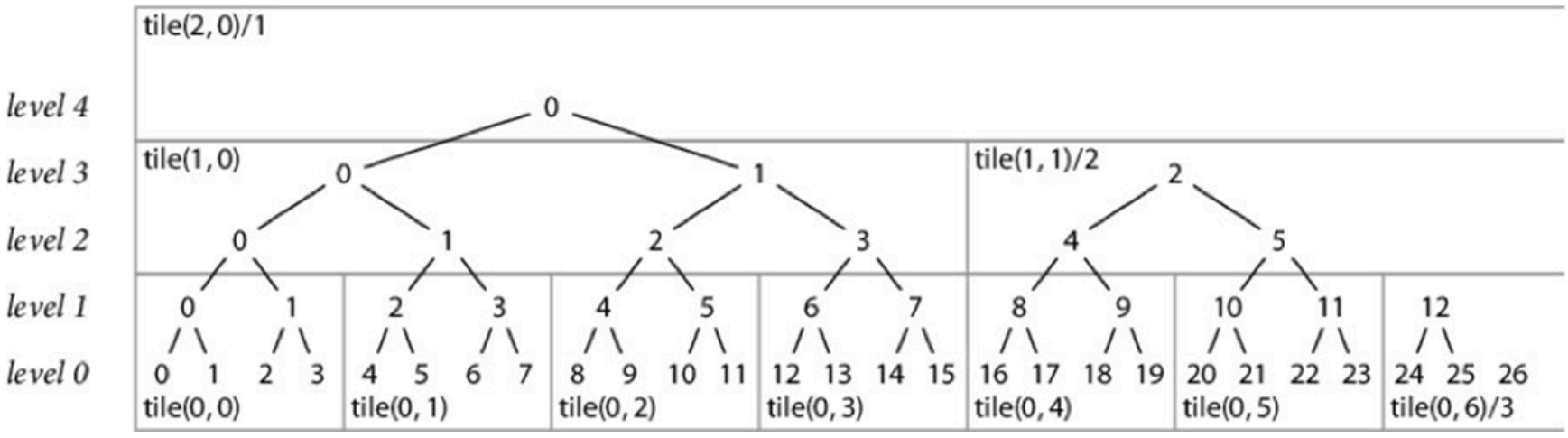

Figure 7. Tiling: storing a tree as a collection of subtrees..

Storage of each tile requires storing only the bottom hashes: the upper rows can be recomputed “on the fly” by hashing lower ones when the tile is loaded into memory. Node updates affect one or more tiles, so in-memory caching is used to increase emciency. After updates are calculated, the cache is flushed to storage, so each affected tile is only written once.

Tiles improve reads too, because it is likely that several nodes traversed in Merkle paths for proof are part of the same tile. The number of tiles involved in a path through a large tree from leaves to root is also bounded.

More details can be found in “Transparent Logs for Skeptical Clients“ under the “Tiling a Log” chapter and in Trillian “Storage Design Notes”.

3. Cryptographic design: Certificate Transparency background

“Rolling your own crypto” is practically always a bad idea. Especially when many stakeholders should trust a system: presenting understandable and familiar technology is more audit-friendly than suggesting myGaru’s partners review a home-brewed Merkle Tree-based data structure implementation.

That’s why we’ve picked a cryptosystem that matches our use-case sumciently: append-only audit log for many distributed write-only components (loggers) and a few read-only verifiers.

Append-only audit logs are used in systems that are required to prove their own transparency and accountability. Below we provide quotes from papers that describe Contour and Certificate Transparency technologies.

Extract from Contour: A Practical System for Binary Transparency paper:

“One of the technical settings in which the idea of transparency has been most thoroughly — and successfully — deployed is the issuance of X.509 certificates. This is partially due to the nature of these certificates (which are themselves intended to be globally visible), and partially to the many publicized failures of major certificate authorities (CAs). Despite the differences in implementations, many of these systems share a fundamentally similar architecture: after being signed by CAs, certificates are stored by log servers in a globally visible append-only log; i.e., in a log in which entries cannot be deleted without detection.

Clients are told to not accept certificates unless they have been included in such a log and to determine this they rely on auditors, who are responsible for checking the inclusion of the specific certificates seen by clients. Because auditors are often thought of as software running on the client (e.g., a browser extension), they must be able to operate emciently. Finally, in order to expose misbehavior, monitors (inemciently) inspect the certificates stored in a given log to see if they satisfy the rules of the system.

To prevent clients from accepting bad certificates, such systems thus rely on monitors to expose them. Because auditors are the ones communicating with the client, however, to achieve this property an additional line of communication is needed between the auditor and monitor in the form of a gossip protocol. In such a protocol, the auditor and monitor periodically exchange information on their current and previous views of the log, which allows

them to detect whether or not their views are consistent, and thus whether or not the log server is misbehaving by presenting “split” views of the log.

If such attacks are possible, then the accountability of the system is destroyed, as a log server can present one log containing all certificates to auditors (thus convincing it that its certificates are in the log), and one log containing only “good” certificates to monitors (thus convincing them that all participants in the system are obeying the rules).”

Note: in a context of this paper, “client” is a software that connects to a server via TLS (browser, mobile and desktop applications); “auditor” is a software that helps client to validate TLS certificates; “monitor” is a software that verifies TLS certificates in Certificate Transparency log. Client, auditor and monitor are roles (functions) of software. Client software is operated by a real person who connects to the internet and wants to make sure that their connection is secure (has valid TLS certificate). Please refer to the detailed definitions in section 4.1 Participants.

Extract from RFC 6962 Certificate Transparency:

“Certificate transparency aims to mitigate the problem of misissued certificates by providing publicly auditable, append-only, untrusted logs of all issued certificates. The logs are publicly auditable so that it is possible for anyone to verify the correctness of each log and to monitor when new certificates are added to it. The logs do not themselves prevent misissue, but they ensure that interested parties (particularly those named in certificates) can detect such misissuance. Note that this is a general mechanism, but in this document, we only describe its use for public TLS server certificates issued by public certificate authorities (CAs).

Each log consists of certificate chains, which can be submitted by anyone. It is expected that public CAs will contribute all their newly issued certificates to one or more logs; it is also expected that certificate holders will contribute their own certificate chains. In order to avoid logs being spammed into uselessness, it is required that each chain is rooted in a known CA certificate. When a chain is submitted to a log, a signed timestamp is returned, which can later be used to provide evidence to clients that the chain has been submitted. TLS clients can thus require that all certificates they see have been logged. Those who are concerned about misissue can monitor the logs, asking them regularly for all new entries, and can thus check whether domains they are responsible for have had certificates issued that they did not expect. What they do with this information, particularly when they find that a misissuance has happened, is beyond the scope of this document, but broadly speaking, they can invoke existing business mechanisms for dealing with misissued certificates. Of course, anyone who wants can monitor the logs and, if they believe a certificate is incorrectly issued, take action as they see fit.

Similarly, those who have seen signed timestamps from a particular log can later demand proof of inclusion from that log. If the log is unable to provide this (or, indeed, if the corresponding certificate is absent from monitors' copies of that log), that is evidence of the incorrect operation of the log. The checking operation is asynchronous to allow TLS connections to proceed without delay, despite network connectivity issues and the vagaries of firewalls. The append-only property of each log is technically achieved using Merkle Trees, which can be used to show that any particular version of the log is a superset of any particular previous version. Likewise, Merkle Trees avoid the need to blindly trust logs: if a log attempts to show different things to different people, this can be emciently detected by comparing tree roots and consistency proofs. Similarly, other misbehaviors of any log (e.g., issuing signed timestamps for certificates they then don't log) can be emciently detected and proved to the world at large.”

Note: in the context of this RFC, “client” is a software that connects to a server via TLS (browser, mobile and desktop applications) or connects to Certificate Transparency log to validate certificate issuance. Client software is operated by a real person who connects to the internet and wants to make sure that their connection is secure (has valid TLS certificate). Please refer to the detailed definitions in section 5. Clients of RFC 6962.

4. Applying Certificate Transparency approach to the myGaru business challenge

The particularity of the myGaru case is the following:

- We use custom transactions instead of certificates (part of a certificate's signature verification is replaced by a transaction correctness verification).
- Considering that the Monitor’s check can be successful, but the Auditor’s one — not (and vice versa), and since it is currently undefined who of the stakeholders will be granted Monitor/Auditor roles, the verification process should include both these checks along with the End Users’ check.

The mentioned above particularities affect only the log verification process while keeping the log creation identical to the CA transparency case.

At first, each stakeholder entity should be able to act as a Monitor and as an Auditor and perform both checks. Moreover, a so-called gossip protocol is to be implemented between the Monitor nodes to prevent split-view attacks.

The core mechanism behind the myGaru transparency solution is the exchange of specific cryptographic data provided by the Audit Log service between all Audit Log entities (some act as Monitors and some as Auditors), and then the comparison of data to detect differences. The custom logic of transaction evaluation is a subject of additional business checks.

Another requirement is to verify that provided transactions do correspond with the End User's identity, but this is out of scope for this document.

The current solution preserves the following security guarantees:

- emcient proving of log entry inclusion (End User's check);
- emcient proving of integrity and append-only log properties (Auditor's check);
- enumerating of all entries, looking for forged entries (Monitor's check);
- protection against split-view attacks.

We use the Trillian project as a backend for implementing Merkle trees. This append-only log server uses the Merkle tree structure as a core component, so below we list common definitions and terms useful for further specifying the details of our solution.

5. Implementing Trillian-based audit log

5.1. Architecture and components of Trillian-based audit log system

A low-level architecture of the system that generates and verifies audit logs has the following components:

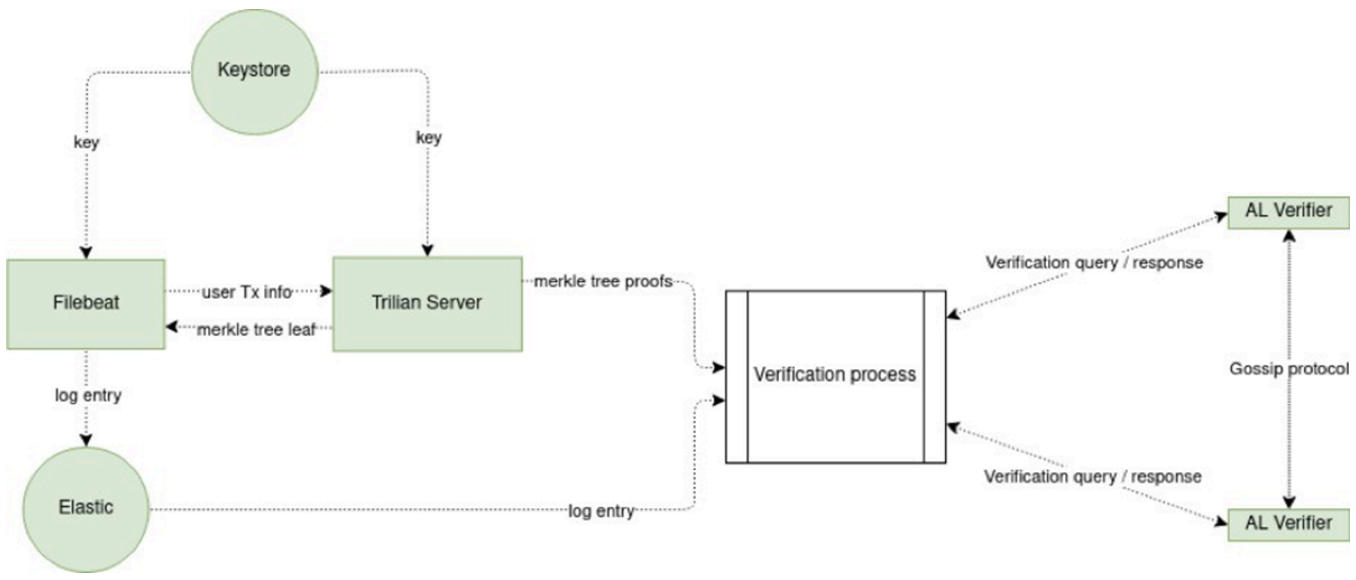

Figure 8. A low-level architecture of the myGaru audit log system.

Filebeat. Filebeat is responsible for (1) generating log entries, based on the End User's transactions (clicks on an advertisement) and interactions between services, (2) digitally signing them, and (3) pushing them to the Elastic server (collector of logs). Filebeat stores its cryptographic keys in Keystore. Filebeat communicates with Trillian in order to obtain particular cryptographic markers that should be part of each log entry.

Elastic server (ELK). Elastic server stores log entries. A party, interested in the verification process, is able to observe log entries and correspondent cryptographic markers by accessing Elastic.

Keystore. Keystore stores cryptographic keys and provides them for Trillian and Filebeat.

Trillian server. Trillian is a service that maintains the Merkle Tree structure — a base for append-only log with security guarantees defined above.

AL Verifier. Audit Log verifier is a service that incorporates the roles of Monitor, Auditor, and End User, and that is able to perform verifications according to the security guarantees. Stakeholders and End Users have access to AL Verifier and are able to perform checks (Monitor check, Auditor check and End User check, accordingly).

We simplify low-level architecture for easier understanding:

- There can be many Filebeat instances controlled by different entities (DSP, Telecom, Partners) which process sensitive data or make decisions related to processing.
- There can be many Trillian and Elastic instances, but a single myGaru entity will control them all.
- There can be many Keystore instances controlled by different entities (typically one for Filebeat and one for Trillian).
- There can be more than two AL In such a case, the Gossip protocol will turn into a Consensus protocol between multiple entities.

5.2. Inclusion of transaction info

Filebeat generates and signs a custom transaction with its private key whenever any operations with sensitive data have to be logged. Then Filebeat asks Trillian to include transaction into Merkle Tree, and Trillian returns a particular response that could be of two forms (both signed by Trillian private key): inclusion proof directly or so-called Signed Timestamp (a promise to incorporate the transaction info in the Merkle Tree within a fixed amount of time called Maximum Merge Delay (MMD)).

If the second option is used, the AL Verifier can obtain inclusion proof by requesting it from Trillian after MMD interval. The “privacy vs system complexity” tradeoff is relevant in this case (refer to the chapter “Inclusion Proofs vs. Promises” of Transparent logging guide by Trillian).

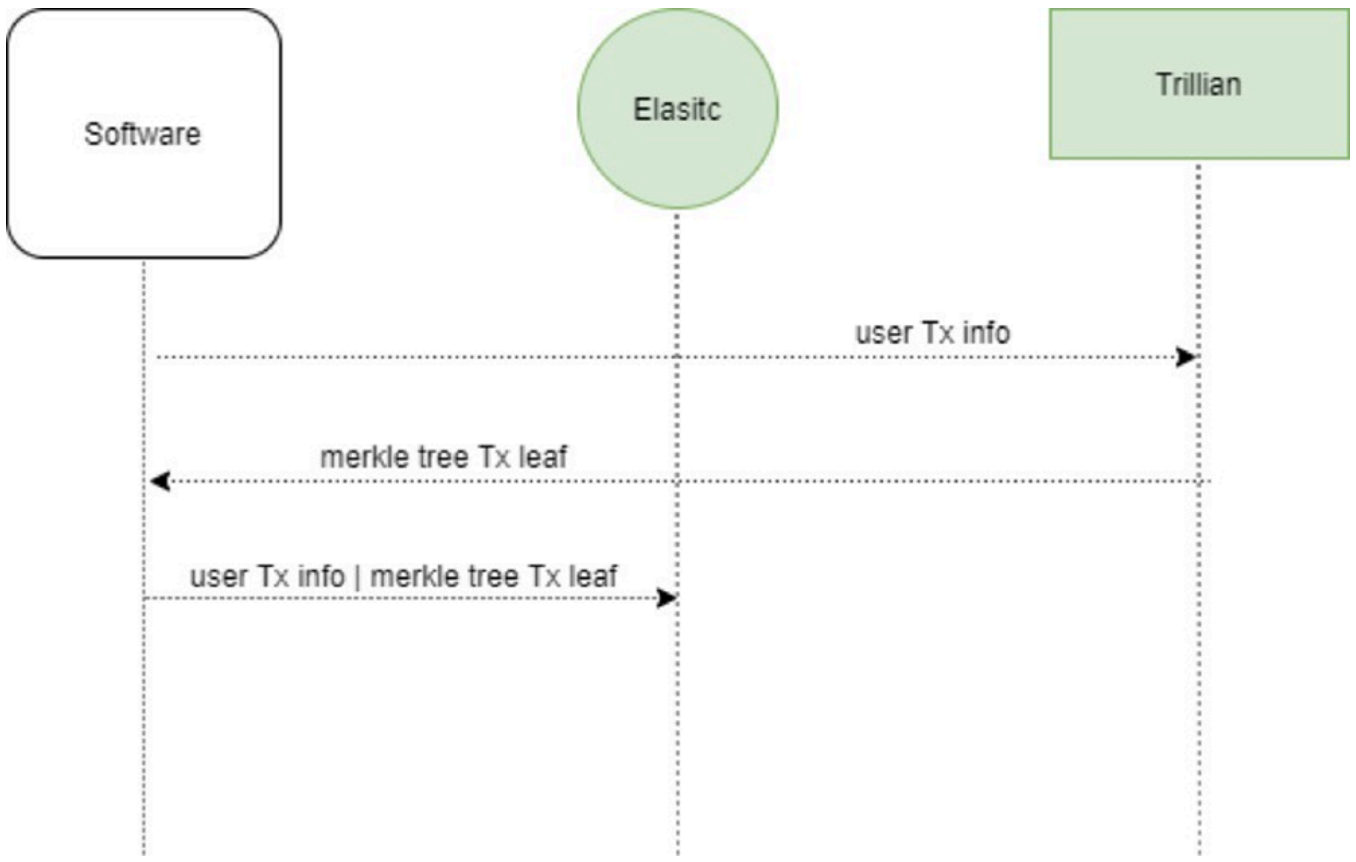

Figure 9. All software components of the myGaru system log operations on sensitive data via Filebeat, Trillian and Elastic.

5.3. End User’s check: verification of inclusion of particular transaction

A stakeholder is interested to check if a particular End Users' transaction has been logged. For this purpose, the AL Verifier extracts the End Users’ transaction info from Elastic and requests the root of Merkle Tree and siblings for this transaction. The root of the tree and set of siblings is sumcient information to check if a particular transaction indeed is presented in the append-only log.

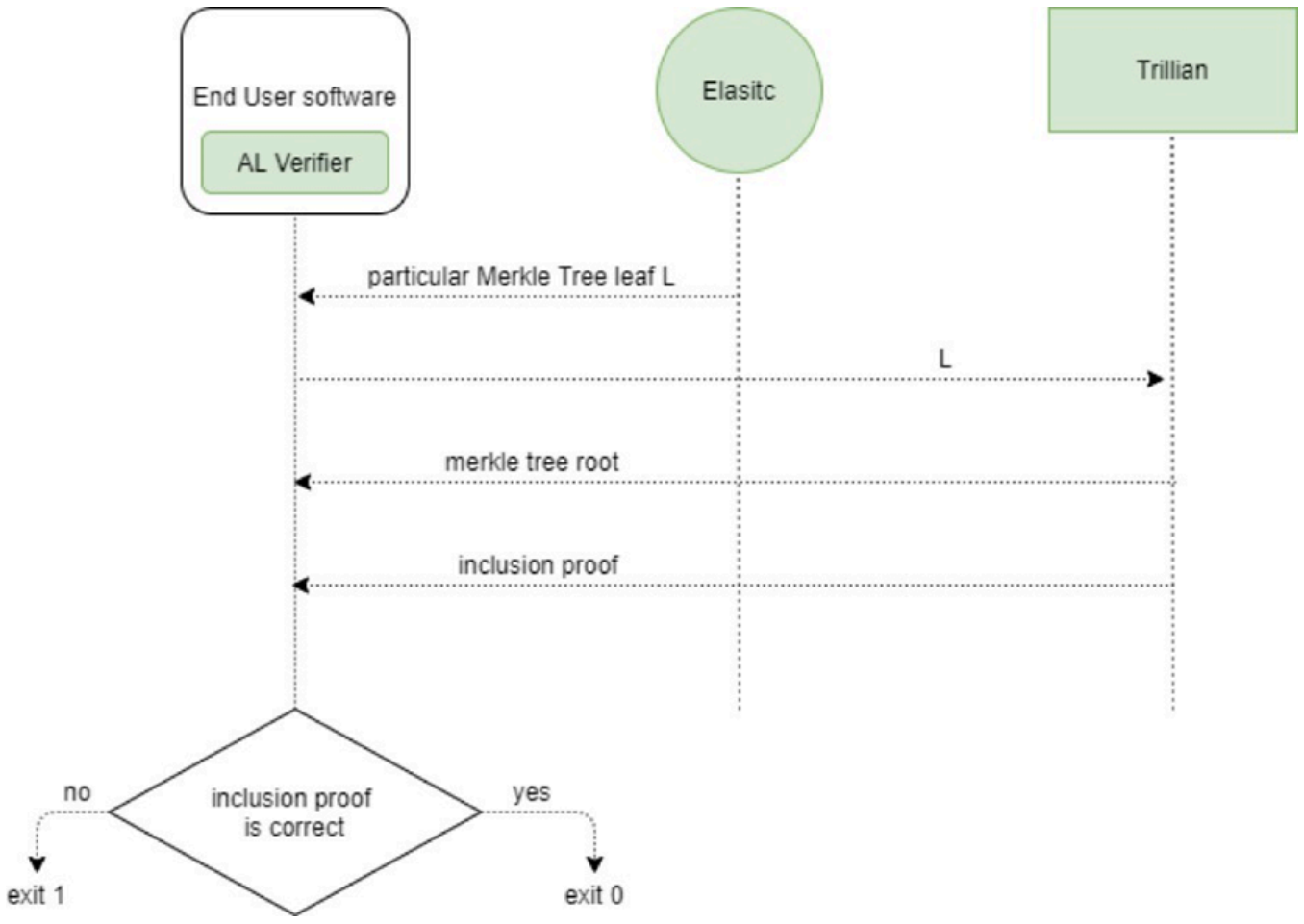

Figure 10. The inclusion proof scheme: a verification that End Users' transaction exists in the tree.

5.4. Auditor's check: verification of append-only property

An Auditor is interested to check if the Trillian server behaves honestly and transparently in the sense that the log grows consistently (e.g. if all previous transactions are presented in the new version of the log).

For this purpose the Auditor on a permanent basis requests a root of the Merkle Tree and consistency proof from Trillian. Once consistency verification is finished, the requested root is stored in the Auditor's local storage, and will participate in further subsequent verifications. The Auditor's local storage can be empty only if this is a first-time verification. Using the consistency property of the Merkle tree structure, the Auditor can be convinced that the log is consistent and append-only.

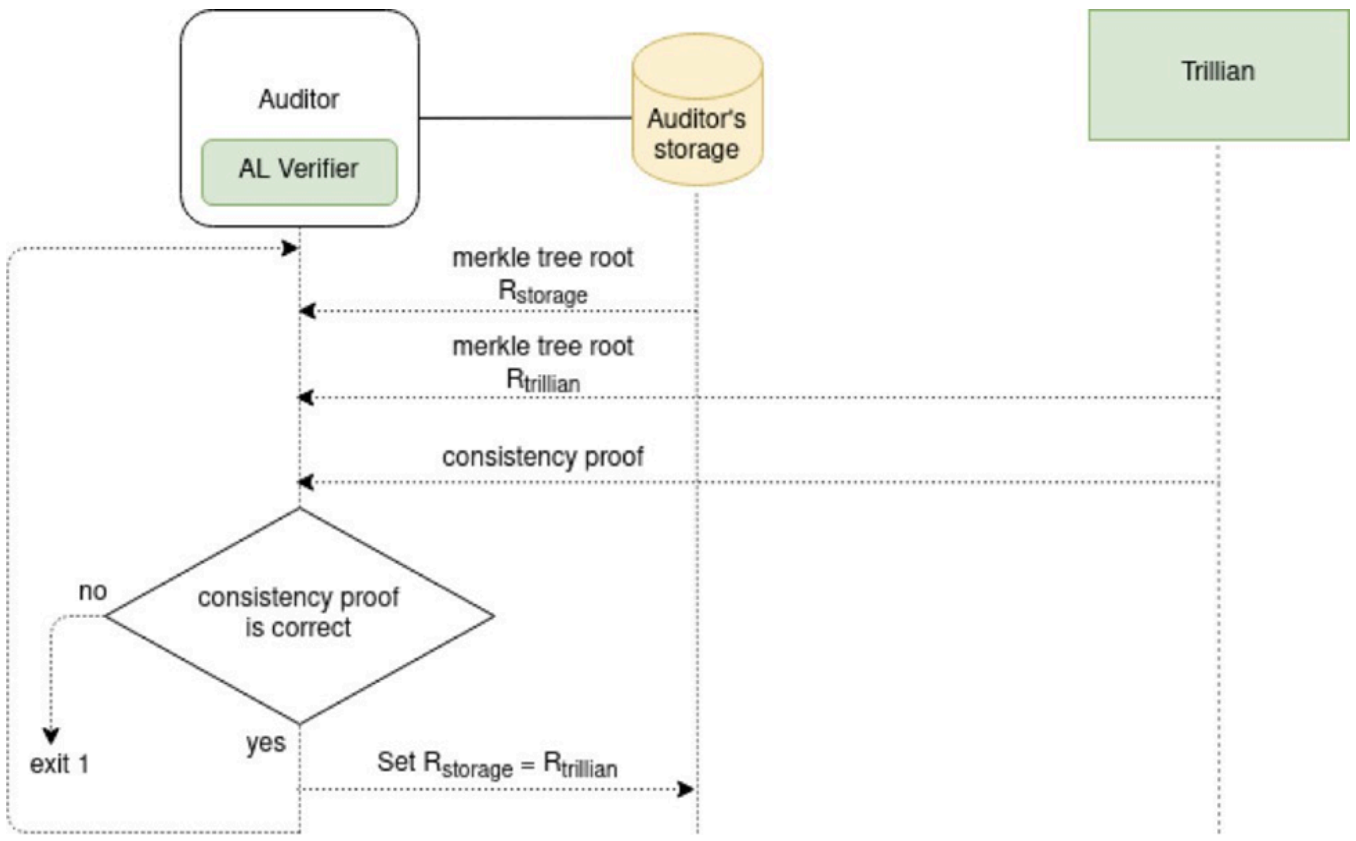

Figure 11. The consistency proof scheme: a verification that the audit log is append-only and consistent

Obviously, consistency proof is requested to check consistency Rstorage ⇔ Rtrillian.

5.5. Monitor's check: evaluation of newly observed transactions

A Monitor is interested to check if the Trillian server behaves honestly and transparently in the sense that each recently added transaction is valid according to business rules.

For our case, those rules are not yet completely specified.

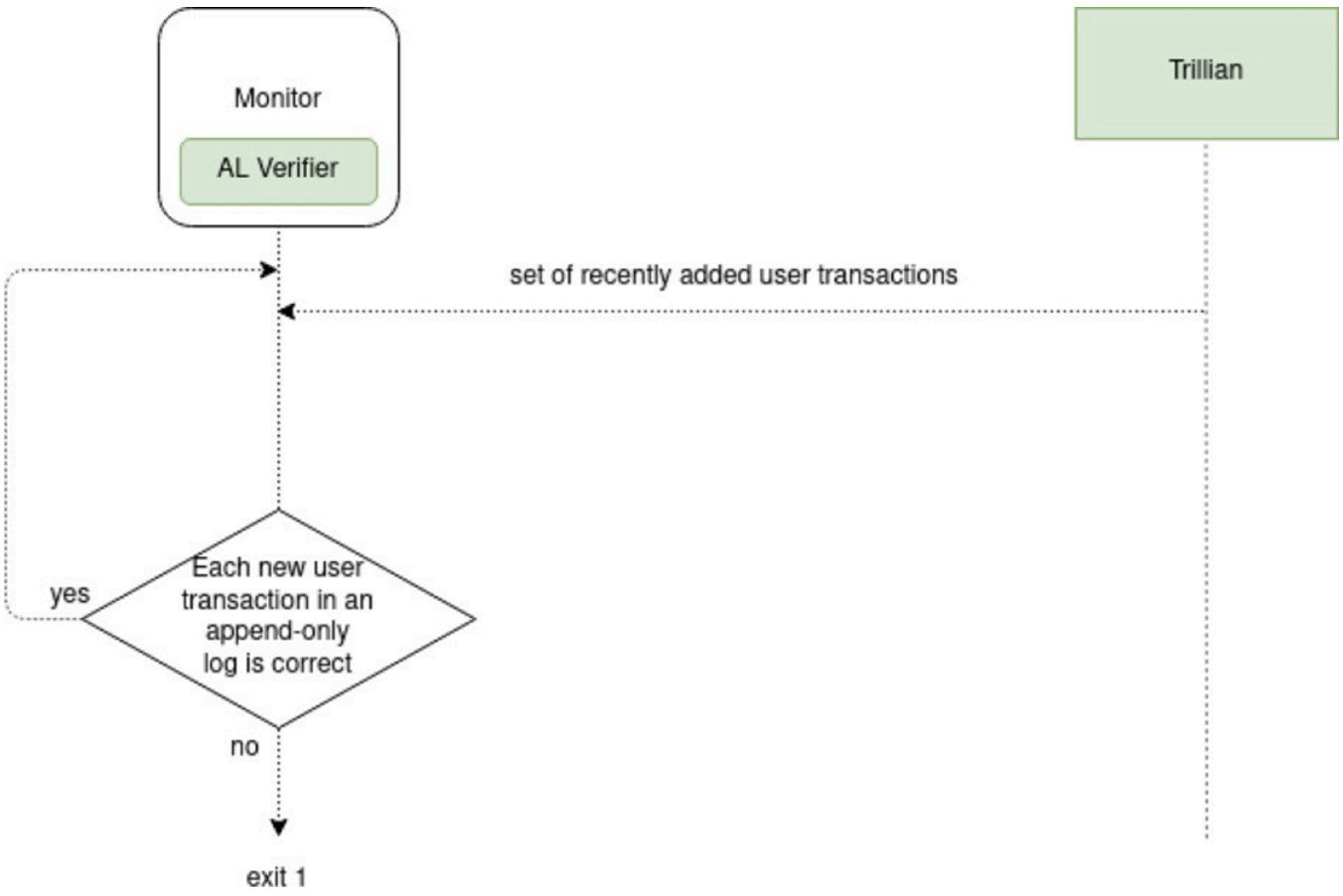

Figure 12. The validity proof scheme: a verification that each recently added transaction is valid according to the business rules.

5.6. Gossip protocol

Gossip protocol is an important part of the verification process and a key supplement to the transparency and security of the whole system.

For simplicity, we specify that the protocol is executed between two participants. For more than two participants, gossip protocol is changed to a consensus protocol, which is out of the scope of the current specification. We only note that some blockchain-like projects (Naivechain) are considered a consensus scheme.

The core principle behind the gossip protocol is outlined below.

Two AL Verifiers should exchange the following information between each other:

- roots of the Merkle tree;
- N inclusion proofs for randomly selected Merkle tree leaves.

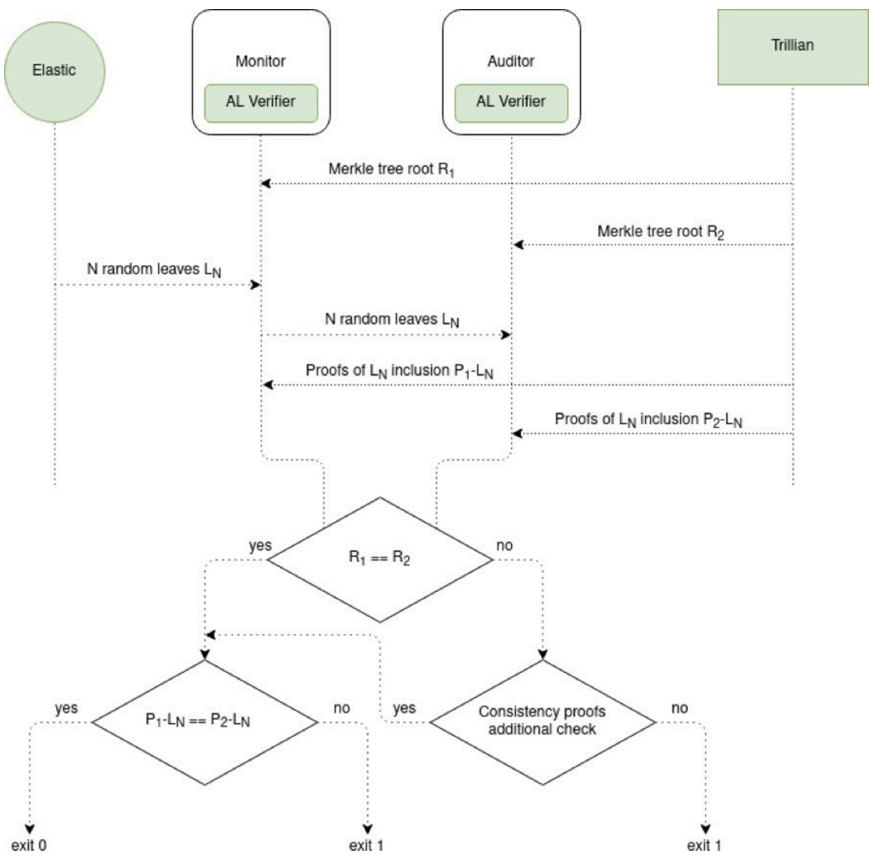

Figure 13. Gossip protocol between two AL Verifiers.

If roots are different, then consistency proof of the Merkle tree should be additionally evaluated as a subprotocol:

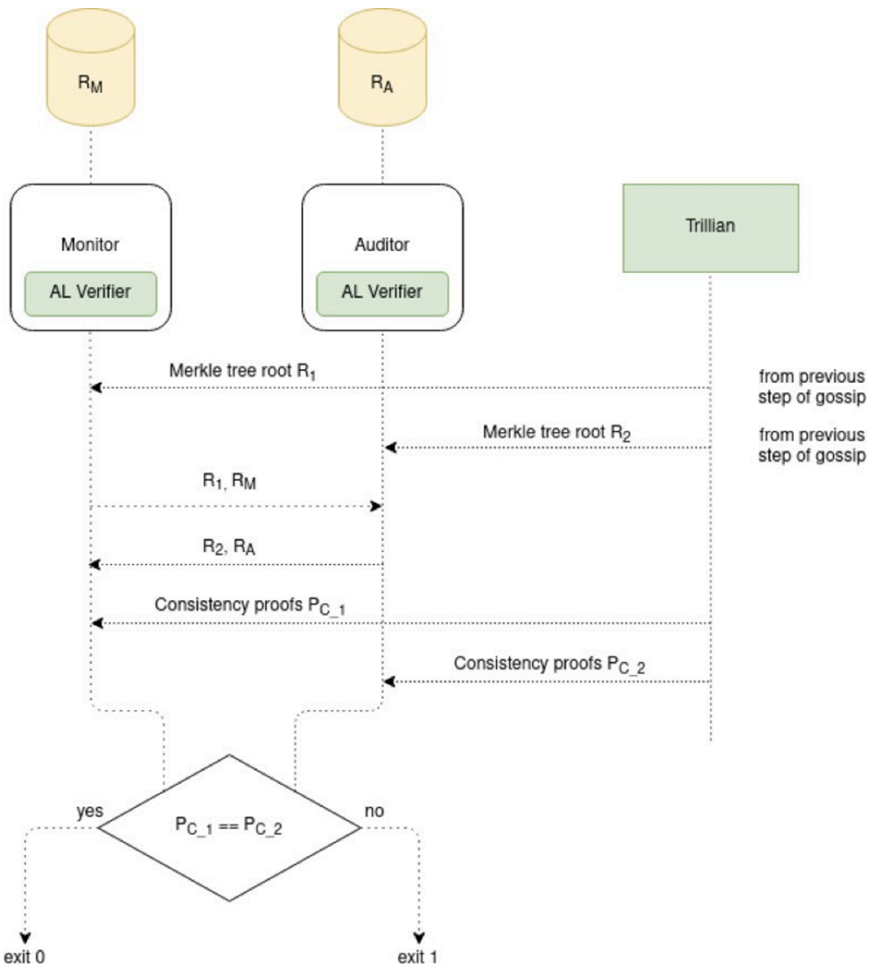

Figure 14. Subprotocol of a gossip protocol in a case when AL Verifiers have different roots of the Merkle tree.

Note here, that R_m, R_a — are the cached Merkle tree roots obtained from the Trillian server during the Auditor's check of each AL Verifier (based on implementation assumption that AL Verifier behaves as a Monitor and as an Auditor).

During this protocol, each AL Verifier obtains R_m, R_a, R_1, R_2. Sets of consistency proofs P_{c_1} and P_{c_2} requested from Trillian are used to check six following consistencies by each AL Verifier:

- $R_m \Leftrightarrow R_a$;
- $R_m \Leftrightarrow R_1$;
- $R_m \Leftrightarrow R_2$;
- $R_a \Leftrightarrow R_1$;
- $R_a \Leftrightarrow R_2$;
- $R_1 \Leftrightarrow R_2$.

Using this consistency verification, both a Monitor and an Auditor are certain that they possess a common collection of logs represented over the Merkle Tree.

If the values above are identical, then the Trillian server is not forged and the verification process (either Monitor's check or Auditor's check or any of the End User’s checks) can be trusted for each of the AL Verifiers.

The result of gossip protocol should be properly handled by interested parties according to the specific business rules. In the Certificate Transparency field, gossiping is yet undefined (only experimental RFC “Gossiping in CT” exists), because TLS handshake contains multiple timestamps from log servers that represent different organizations, which are unlikely to maliciously collude regarding the particular certificate. In contrast, the gossip protocol is essential in the X.509-PKI domain, because typically the X.509 certificate is presented by client-side software which makes it easier to manipulate.

In our setting, gossip protocol is essential, since the log server is a single entity maintained within the myGaru organization. We will update this paper with Gossip protocol specification once it has been validated against business rules in practical implementation.

6. Integrating a Trillian-based audit log

Implementing a logging solution is not enough for a truly auditable system. The next step is integrating it across services and ensuring that no significant activity goes unlogged.

6.1. Current level: always-loggable choke points

myGaru's ecosystem is built around dozens of software components in separate private cloud locations and data centres. The system is designed to ensure sound processes: there are choke points that enforce security controls no-matter-what. For instance, accessing encrypted sensitive data is only possible through a service that holds the cryptographic keys — thus, logging access at this point ensures that any sensitive data access has been logged.

Strategically placing these "un-bypassable" data flow choke points and adding logging there allows us to ensure that every access activity has been recorded.

6.2. Future work: business logic validation

As myGaru's solution becomes more stable in its core, we're looking into formal representation of the business processes that touch any sensitive data — be they with PII or anonymized datasets.

Based on it, we're producing a way to validate events against pre-defined finite-state machines ("a logical trajectory of events that are valid to happen with sensitive data"). We will keep a separate log that marks every event as "matched against valid business logic" vs "looks anomalous" to ensure that Monitors can discover anomalies aside from verifying that the log is honest.

References and related materials

- RFC 6962: Certificate Transparency https://datatracker.ietf.org/doc/html/rfc6962
- Attacking Merkle Trees With a Second Preimage Attack / Dean Jerkovich ttps://flawed.net.nz/2018/02/21/attacking-merkle-trees-with-a-second-preimage-attack/
- Trillian: Storage Design Notes / Martin Smith ttps://github.com/google/trillian/blob/master/docs/storage/storage.md
- Transparent Logs for Skeptical Clients / Russ Cox https://research.swtch.com/tlog
- Contour: A Practical System for Binary Transparency / Mustafa Al-Bassam and Sarah Meiklejohn https://arxiv.org/pdf/1712.08427.pdf
- Transparent Logging: A Guide https://github.com/google/trillian/blob/master/docs/TransparentLogging.md
- Gossiping in CT https://datatracker.ietf.org/doc/html/draft-ietf-trans-gossip-03

myGaru