Keyless Signatures' Infrastructure: How to Build Global Distributed Hash-Trees

Ahto Buldas[1,2], Andres Kroonmaa[1], and Risto Laanoja[1,2]

[1] GuardTime AS, Tammsaare tee 60, 11316 Tallinn, Estonia
[2] Tallinn University of Technology, Raja 15, 12618 Tallinn, Estonia

Abstract. Keyless Signatures Infrastructure (KSI) is a globally distributed system for providing time-stamping and server-supported digital signature services. Global per-second hash trees are created and their root hash values published. We discuss some service quality issues that arise in practical implementation of the service and present solutions for avoiding single points of failure and guaranteeing a service with reasonable and stable delay. Guardtime AS has been operating a KSI Infrastructure for 5 years. We summarize how the KSI Infrastructure is built, and the lessons learned during the operational period of the service.

1 Introduction

Keyless signatures are an alternative solution to traditional PKI signatures. The word *keyless* does not mean that no cryptographic keys are used during the signature creation. Keys are still necessary for authentication, but *the signatures can be reliably verified without assuming continued secrecy of the keys*. Keyless signatures are not vulnerable to key compromise and hence provide a solution to the long-term validity of digital signatures. The traditional PKI signatures may be protected by timestamps, but as long as the time-stamping technology itself is PKI-based, the problem of key compromise is still not solved completely.

Keyless signatures are a solution to this problem. In a keyless signature system, the functions of signer identification and of evidence integrity protection are separated and delegated to cryptographic tools suitable for those functions. For example, signer identification may still be done by using asymmetric cryptography but the integrity of the signature is protected by using keyless cryptography—the so-called *one-way collision-free hash functions*, which are public standard transformations that do not involve any secret keys.

Keyless signatures are implemented in practice as *multi-signatures*, i.e. many documents are signed at a time. The signing process involves the steps of:

1. *Hashing*: The documents to be signed are hashed and the hash values are used to represent the documents in the rest of the process.
2. *Aggregation*: A global temporary per-round hash tree is created to represent all documents signed during the round. The duration of rounds may vary but is set to one second in the working solution.

H. Riis Nielson and D. Gollmann (Eds.): NordSec 2013, LNCS 8208, pp. 313–320, 2013.

3. *Publication*: The root hash values of the per-round aggregation trees are collected into a perpetual hash tree (so-called *hash calendar*) and the root hash value of that tree is published as a trust anchor.

To use such signatures in practice, one needs a suitable *Keyless Signatures' Infrastructure (KSI)* analogous to PKI for traditional signature solutions. Such an infrastructure consists of a hierarchy of aggregation servers that, in co-operation, create the per-round global hash trees. First layer aggregation servers, so-called *gateways*, are responsible of collecting requests directly from clients, and every aggregation server receives requests from a set of lower level servers, hashes them together into a hash tree, and sends the root hash value of the tree as a request to higher-level servers. The server then waits for the response from a higher-level server and by combining the received response with suitable hash chains from its own hash tree responds to lower-level servers.

In this paper, we discuss some service quality and availability issues that arise when maintaining this tree in practice and describe solutions to overcome them. The implementation avoids single points of failure and guarantees reasonable and stable service latency. Guardtime AS has been operating a KSI Infrastructure for 5 years—sufficiently long time to draw some conclusions about the availability, scalability and practical lessons learned during the operational phase. This paper summarizes how the KSI Infrastructure is built, its main components and the operational principles. We provide a brief overview of the security aspects of the service, including design decisions that minimize possible risks.

2 Hash Trees and Hash Calendars

Hash Trees: Hash-tree aggregation as a technique was first proposed by Merkle [6] and first used for digital time-stamping by Haber et al [5]. Hash-tree time-stamping uses a one-way hash function to convert a list of documents into a fixed length digest that is associated with time. User sends a hash of a document to the service and receives a *signature token*—a proof that the data existed at the given time and that the request was received through a specific access point. All received requests are *aggregated* together into a large hash tree; and the top of the tree is fixed and retained for each second (Fig. 1). Signature tokens contain data for reconstructing a path through the hash tree—starting from a signed hash value (a leaf) to the top hash value. For example, to verify a token y in the place of x_1 (Fig. 1), we first concatenate y with x_1 (part of the signature token) and compute a hash value $y_2 = h(x_1 \mid y)$ that is used as the input of the next hash step, until we reach the top hash value, i.e. $y_3 = h(y_2 \mid x_{34})$ in the example case. If $y_3 = x_{top}$ then it is safe to believe that y was in the original hash tree.

Hash Calendar: Root hash values for each second are *linked* together, into a globally unique hash tree called a *hash calendar*, so that new leaves are added only to one side of the tree. Time value is encoded as the *shape* of the calendar the modification of which would be evident to other users. The top hash of the calendar is periodically published in widely witnessed media.

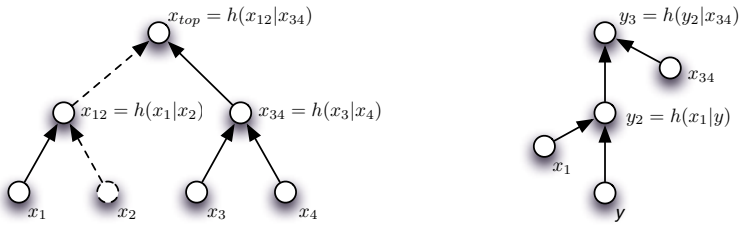

Fig. 1. Computation of a hash tree (left), and verification of y at the position of x_2

There is deterministic algorithm to compute top of the linking hash three, giving us distinct top level hash value at each second. Also there is an algorithm to extract time value from the shape of the linking hash tree for each second, giving us a hard-to-modify time value for each issued token.

Security Against Back-Dating: Security against back-dating means that malicious servers must be unable to add new fresh requests to already published hash trees. It has been shown [4,3,2] that if the hash function is secure in ordinary terms (one-wayness, collision-resistance, etc.) and the aggregation tree is of limited size, then the scheme is indeed secure against back-dating.

3 System Architecture

An Application (Fig. 2) computes a hash of the document that is going to be signed and sends a request to a Gateway—a server that delivers the service to end-users. Gateway aggregates the requests received during an aggregation cycle and sends its top hash value as a request to the upstream aggregation cluster. The request is aggregated through multiple layers of Aggregator servers, and the globally unique top hash value is created by the Core cluster. The response (that consists of verifiable hash tree paths) is sent immediately back through the aggregation layers. The top hash values for each second are collected to Calendar Archive and distributed through the *Calendar Cache* layer to the Extender service, usually co-located with the Gateway host. Client applications use the Extender service for the verification of signatures.

Aggregation Network: An *aggregator* is a system component that builds hash trees from all incoming requests and passes root hash values to upstream system components. Aggregators work in rounds of equal duration. The requests received during a round are aggregated into the same hash tree. After receiving a response from an upstream component, an aggregator immediately delivers the response to all child aggregators together with hash paths of its own tree. Subsequent responses from upstream components for the same round are ignored. The aggregation tree is split to four layers, and an aggregation infrastructure was built so that the top layer is close to the Core cluster (see the next section), two intermediate layers provide geographic scale. The bottom layer is bundled with Gateways and hosted typically at the end user premises. Each downstream

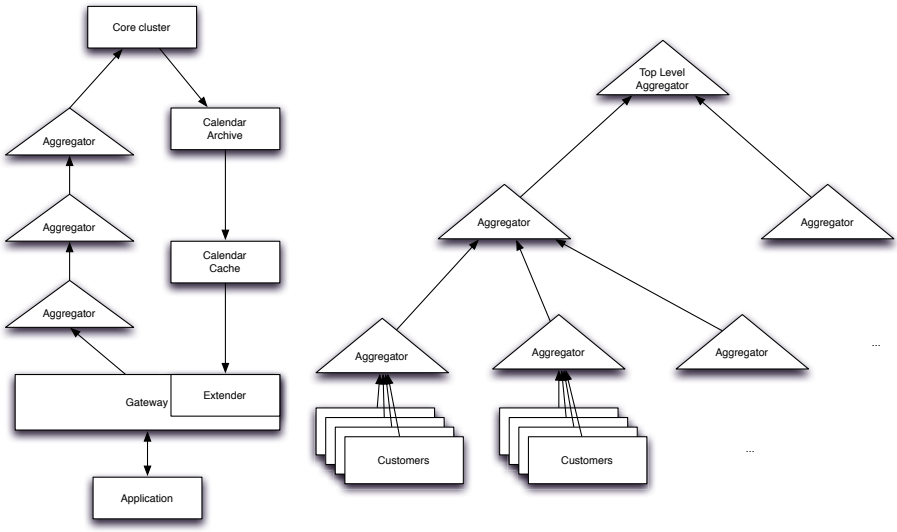

Fig. 2. High-level system architecture and the aggregation network

client or aggregator has its reserved spot in the hash tree—this allows to prove which server was involved in the creation of a particular signature token.
The aggregation tree scales well. In order to double the system capacity we have to add only one hash value to the signature token. Current hash-tree depth is fixed at 50 steps, giving us theoretical maximum capacity of $2^{50} \approx 10^{15}$ signatures per second. This initial configuration is believed to cover possible signature needs for the foreseeable future. Each gateway and aggregation server generates *constant* upstream network traffic which does not depend on the actual load. This isolates the customers, does not leak information about the actual service usage, and provides reasonable denial-of-service attack protection. Also in order to scale up the service it is easy to add resources with linear increase in capacity.

Core Cluster consists of top-level aggregators and is a distributed synchronized machine responsible for producing the hash calendar and propagate it through the aggregation network. The root hash values of the calendar are archived and distributed to verification servers, through guaranteed integrity archiving and "dumb" caching layers. The roots of intermediate aggregation trees are only stored in relevant signature tokens. Top level aggregators guarantee that the time value of the calendar corresponds to the UTC time. Gateways fetch their copies of the Calendar from the cache servers using the HTTP protocol. Local copy of the Calendar data is used for signature token verification.

Gateway: Gateway works as a protocol adapter, accepting requests in application specific formats (RFC3161, OpenKSI) and forwarding them to designated Aggregator(s). In practice, first level of aggregation happens already at a Gateway host, giving us low and predictable communication bandwidth between the Aggregators and Gateway. Gateways host a *Verifier* (or *Extender*)— a signature

verification assistant. Extender has a fresh copy of the calendar, and it builds hash chains from signed hash values to the published hash values. This cannot be done immediately after signing, because part of the calendar is not yet known at the signing time. The hash chains created by the Extender and validated in client APIs. Client applications may store the verified token with full information for re-creation of the hash-chain by creating so-called "extended" token.

4 Availability and Service Quality

To increase the *availability* of the service, single points of failure must be avoided and we have to use redundancy everywhere in the system. Every aggregation server is replaced with a *geographically dispersed cluster* of servers that work in parallel, so that lower-level servers send requests to the whole cluster and will use the first received valid reply. If the availability coefficient of a single server is assumed to be 0.99 (approximate downtime is 3.5 days per year), then a cluster with two servers has availability about 0.9999, assuming total independence of downtime events. The clusters can be enlarged without downtime.

The response time of the service may depend on several characteristics of the network and if no measures are taken may vary considerably. Below we describe how we eliminated the "long tail" of the service response time.

Simplified Approach. The aggregation network is redundant, i.e. it has a cluster of m aggregators instead of one. Every aggregator has a certain aggregation period d (in time units). The larger the aggregation period is, the larger service delay it creates, i.e. a request that receives at random time will be aggregated (i.e. the Merkle tree built, the root hash calculated and sent to the parent cluster) approximately after $d/2$ units of time. This means that every aggregator in the path from a client to the core-cluster adds $d/2$ time units of service delay.

If an aggregation round begins at 0 and ends at d, then a request that arrives at t (in $[0 \dots d]$) will have service delay $d - t$, i.e. the larger t is, the smaller will be service delay. The requests that arrive later (just before the round is closed) have smaller service delays.

The main idea is to adjust the round schedules of the aggregators in the same cluster so that the average delay of requests will be minimal. For example, if we have two aggregators in the cluster both with round length d (in time units) and the round of the second aggregator begins at time $d/2$ (instead of 0), then (as every request is sent to both aggregators), the average service delay is $d/4$ instead of $d/2$. This is because the delay for a request received at t is now the following function $\delta(t) = \frac{d}{2}(1 + \lfloor 2t/d \rfloor) - t = \begin{cases} d/2 - t & \text{if } t \in [0 \dots d/2] \\ d - t & \text{if } t \in [d/2 \dots d] \end{cases}$ and the average value of this function in $[0 \dots d]$ is $d/4$. In general, if we have m aggregators in the cluster, and the round of the i-th aggregator in the cluster begins at time i/m, then $\delta(t) = \frac{d}{m}(1 + \lfloor mt/d \rfloor) - t$ and the average delay is $\frac{d}{2m}$.

This method reduce the service delay by interleaving the aggregation rounds in a cluster. The simplified approach is useful if the delay is almost completely

random, i.e. has a large standard deviation comparable to the duration of aggregation rounds. Such extreme conditions are very rare in practice.

Practical Approach. A network delay between a child aggregator C and a parent aggregator P consists of several components:

- *Propagation delay* caused by the basic fact of physics and which depends on the length of wires between C and P. This delay cannot be eliminated.
- *Serialization delay* caused by global cloud of network routers that choose the paths in the network that are used to send data from C to P.
- *Jitter*. Mostly caused by varying utilization which creates processing queues and causes retransmissions.

All these component-delays create a probability distribution that is not uniform but a rather sharp bell-curve. For example, if we know that 95 per cent of the requests (of C to P) have delays between $25 - 40$ms (milliseconds), then we can adjust the round schedules of C and P, so that their rounds (if they are of equal duration d) begin at t and $t + 40$ms, respectively. This means that 95 per cent of the requests send by P to C have additional delay less than 40ms. Note that in practice, the delay is much smaller than d/m, where m is the number of aggregators in a cluster and d is the aggregation period.

Message flow between the aggregation layers is depicted in Fig. 3. The vertical axis represents layers of the aggregation tree, and the horizontal axis represents time-flow in seconds. Left drawing illustrates the unsynchronized case with two requests, first one being worst case and second one being best case. As request

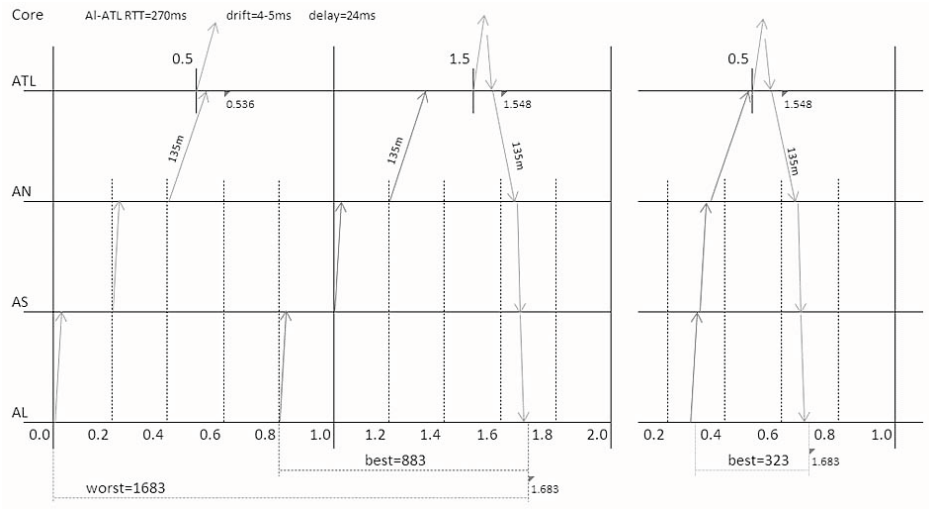

Fig. 3. Message flow through the aggregation layers (AL → AS → AN → ATL → Core and back). Horizontal axis represents the flow of time in seconds.

travels upstream it waits for end of the aggregation round at each layer, and first requests narrowly misses the end of 1-second top level aggregation cycle. Second requests arrives just before the end and response for both requests arrives at the same time. Right drawing depicts the ideal case with synchronized layers.

5 Practical Results

Test Setup: The test was performed during the service expansion to Japan, topologically very distant location from the Core cluster which is distributed between the different jurisdictions in Europe. Service was already extensively tested in the laboratory environment, so that its performance in presence of non-ideal network conditions was already mapped. Also, its latency, when operated within a single continent, had been proven to be satisfactory. We rented some physical and some virtual servers from 5 different service providers based on Tokyo and Nagano. There were also some non-formal testing objectives, like (1) finding set of service providers with least dependent resources, especially ISP peering; (2) testing service quality provided by different sizes of clusters of physical and virtual servers, finding cost effective combination; (3) testing effect of different aggregation periods; (4) testing effect of other system parameters; (5) providing data to draft the Service Level Agreements. Load was generated remotely, measurements were performed at the Gateway host, so that client application to gateway connection did not impact the measurements. Tests were run for 24 hour periods, for at least 3 consecutive days. If possible then ISP-s with worst quality of service were used (they were dropped in production).

Results: The main goal was to improve the service quality, i.e. provide minimal and deterministic latency of the signing service to the end users; and also to find cost effective setup to guarantee reasonable availability. The progress is presented with the before and after response timing histograms in Fig. 4. The left graph depicts the initial real-life signing response timing distribution. Note that there are no failed requests because of the redundancy and automatic retry mechanism on all aggregation layers. Response latency histogram after the synchronization of the aggregation layers and other optimizations like tuning host network stack and Linux kernel parameters is depicted at Fig. 4 (right). Here latency is mostly dictated by the underlying network delays as RTT (round-trip delay) from AN to ATL is approximately 270ms, other network delays are below few milliseconds. Clock drift at all layers is less than 4ms and Core protocol voting time is 48ms. Final optimizations and findings included:

- The lowest latency was achieved by the aggregation period of 200...400ms. We started with 200ms and later reverted to 400ms for less data traffic.
- In redundant clusters, virtual servers are reasonably good. Three virtual servers cost less and provide better service than 2 dedicated physical servers.
- Virtualized servers can have choppy flow of time; it helps to keep local disk IO minimal. For our case it was necessary to set up network logging.
- Although being easier to implement the TCP based network protocol had some unwanted quirks, especially the "*TCP slow start after idle*" feature.

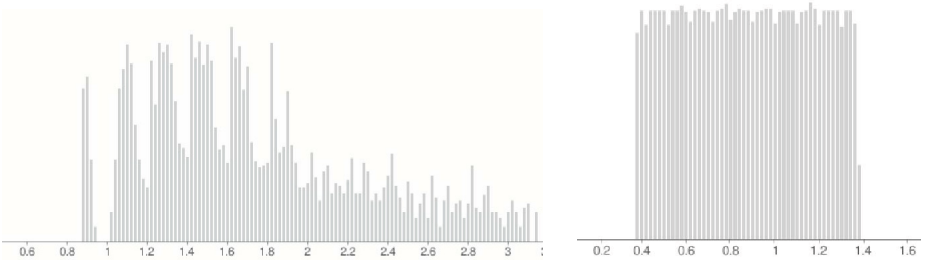

Fig. 4. Real-life response time histograms before and after the optimizations. Vertical axis is the number of samples, horizontal scale is the latency in seconds.

In practice, the synchronization involved setting up reasonably good configuration of Internet-based NTP time synchronization and configuring optimal timing offsets based on measured RTT between the aggregation layers at each Aggregator site. Depending on availability requirements 2 or 3 Aggregation servers in a cluster provided satisfactory results.

References

1. Bayer, D., Haber, S., Stornetta, W.-S.: Improving the efficiency and reliability of digital timestamping. In: Sequences II: Methods in Communication, Security, and Computer Sci., pp. 329–334. Springer, Heidelberg (1993)
2. Buldas, A., Laanoja, R.: Security proofs for hash tree time-stamping using hash functions with small output size. In: Boyd, C., Simpson, L. (eds.) ACISP 2013. LNCS, vol. 7959, pp. 235–250. Springer, Heidelberg (2013)
3. Buldas, A., Niitsoo, M.: Optimally tight security proofs for hash-then-publish time-stamping. In: Steinfeld, R., Hawkes, P. (eds.) ACISP 2010. LNCS, vol. 6168, pp. 318–335. Springer, Heidelberg (2010)
4. Buldas, A., Saarepera, M.: On provably secure time-stamping schemes. In: Lee, P.J. (ed.) ASIACRYPT 2004. LNCS, vol. 3329, pp. 500–514. Springer, Heidelberg (2004)
5. Haber, S., Stornetta, W.-S.: How to time-stamp a digital document. Journal of Cryptology 3(2), 99–111 (1991)
6. Merkle, R.C.: Protocols for public-key cryptosystems. In: Proceedings of the 1980 IEEE Symposium on Security and Privacy, pp. 122–134 (1980)