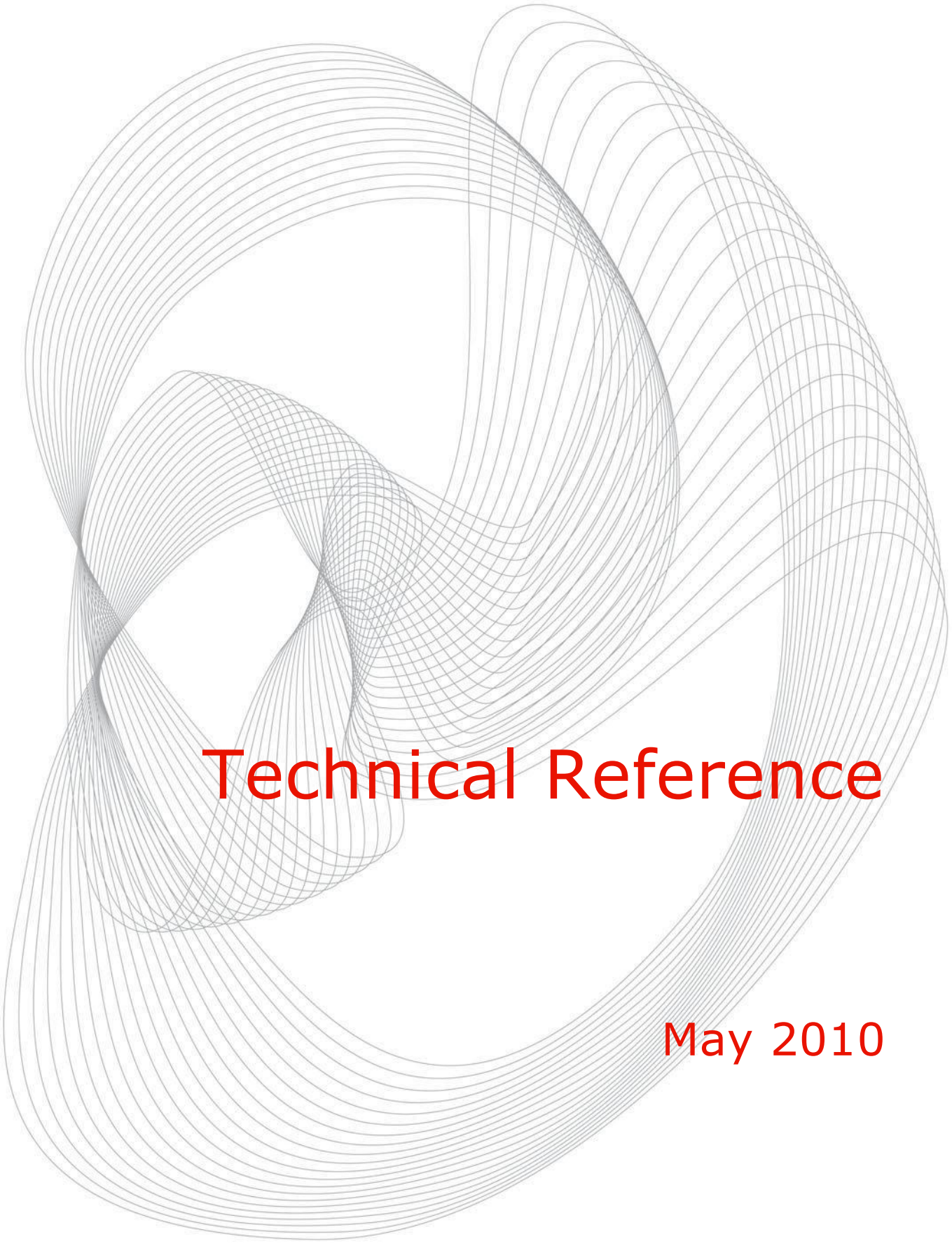
guardtime



Technical Reference

May 2010

guardtime

Contents

1. Overview

The purpose of a cryptographic timestamping system is to provide evidence connecting data to time in a way that allows later verification that the timestamp was indeed issued at the time claimed and the data has not been changed since.

The aim of this document is to provide enough data and references that a reader could understand how the GuardTime timestamping system works. The more detailed file format and algorithm specifications are provided in a detached appendix.

Section 2 gives a short introduction to the timestamping process as whole, followed in Section 3 by an overview of the service architecture and the main components involved.

Section 4 describes the cryptographic building blocks from which the GuardTime timestamping technology is assembled. Of particular importance is the calendar data structure built on top of a Merkle tree.

Sections 5 and 6 explain the timestamping and verification processes in more detail.

Section 7 summarizes the information needed to provide to a third party (such as a court of law or a regulatory authority) for independent proof of data integrity.

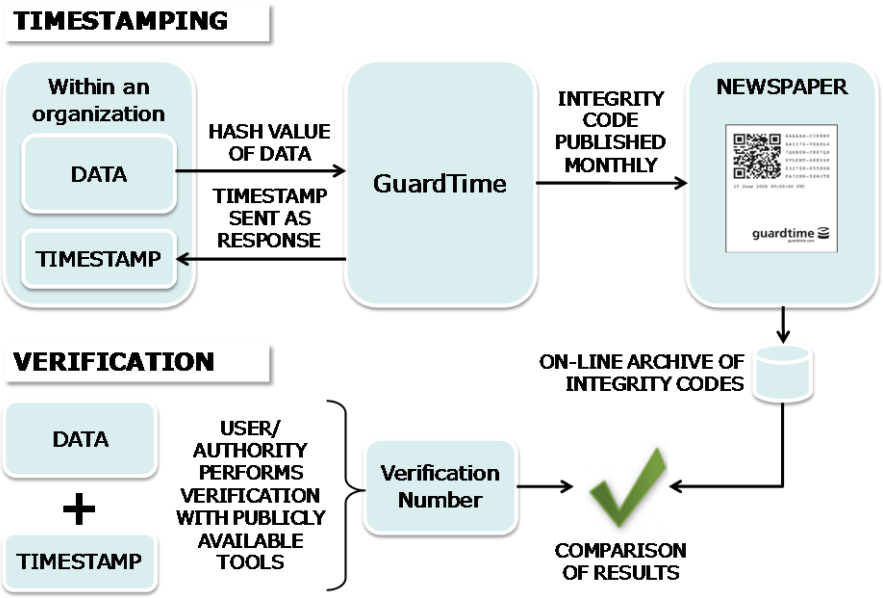
The appendix "GuardTime Technical Reference – Formats and Algorithms" (distributed in a separate document) aims to define all the relevant data formats and verification procedures in detail sufficient for independent creation of verification tools.

2. Introduction

Figure 1 illustrates the overall process of using the GuardTime timestamping service:

- First, the owner of the data computes a hash value of the data and sends it to GuardTime for timestamping. A hash value is like a digital fingerprint: it is unique to the data from which it was computed; it is easy for anyone with the proper tools to verify that the data and its hash value match; and it is impossible to recover the original data from the hash value. Due to the last property, it is safe to timestamp even the most confidential data, as the data itself never leaves the customer's premises.
- The GuardTime service receives the timestamping request, creates a timestamp token for it, and sends the token back to the owner of the data to be archived together with the data.
- Periodically, GuardTime computes and publishes an Integrity Code that summarizes all the timestamps it has issued so far. The Integrity Code is computed in a way that a proof can be constructed for each timestamp showing that the timestamp did, in fact, participate in creation of the Integrity Code, thus demonstrating the integrity and the age of the timestamped data. The Integrity Code is published in widely circulated newspapers for tangible proof and an archive of Integrity Codes is also made available in electronic form for convenience.

Figure 1: Timestamping and verification.



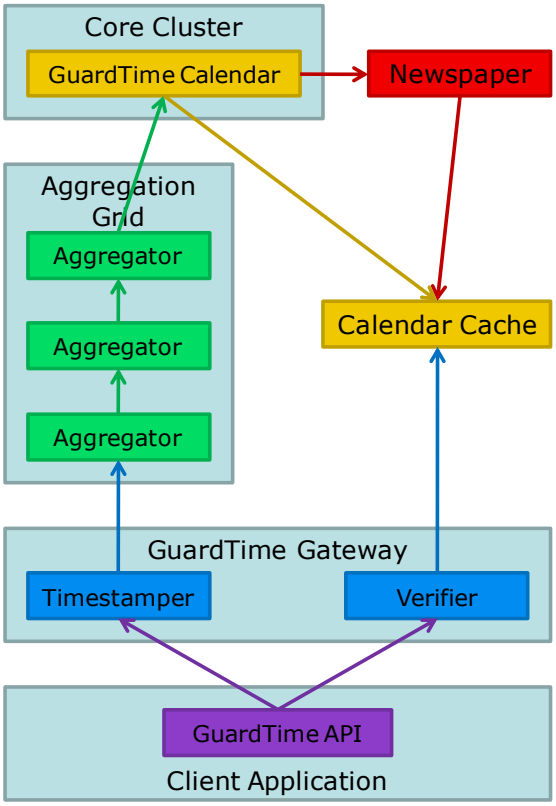
- To prove the integrity of the timestamped data, the owner of the data passes the data and the timestamp to a verification tool. The tool (either one made available by GuardTime or one created by an independent party based on specifications published by GuardTime) computes a Verification Number. If the computed Verification Number matches the published Integrity Code, this concludes the proof that the timestamp is authentic and the data has not been tampered with.
- The verification tool and all relevant specifications are available for independent review, for example by an expert witness appointed by a court of law.

3. Service Architecture

The GuardTime infrastructure (Figure 2) consists of the following components:

- The core cluster maintains the GuardTime calendar. Every second, an aggregate of all the timestamping requests accumulated during the second is added to the calendar and linked to the aggregates of past seconds. The linking information is then distributed down the grid back to the gateway level where it is included in all the timestamps issued for that second.
- The aggregation grid is a hierarchy where each aggregator collects the requests from the layer below and passes only a fixed-size summary on to the next level. This ensures that the load of any server only depends on the number of sub-ordinate servers immediately connected to it, not on the total number of servers in the lower layer, the number of indirect subordinates in more distant layers, or the number of requests issued by those indirect subordinates. This in turn means that the service is almost infinitely scalable: to be able to service more requests, more aggregators and gateways can be added to the system, but the load on the core servers does not increase. There is no bottleneck in the system.
- The calendar cache is responsible for distributing back to the gateway level the calendar data and Integrity Codes needed for constructing the authenticity proofs.

Figure 2: GuardTime service architecture.



- The gateway is the interface between the client application and the GuardTime service. The timestamper module in the gateway is responsible for accepting timestamping requests and returning timestamps. The verifier module supplies the proofs connecting timestamps to the Integrity Codes published in newspapers.

4. Cryptographic Foundations

In this section we review the cryptographic building blocks from which the GuardTime timestamping technology is assembled, before we move on to the signing and verification processes.

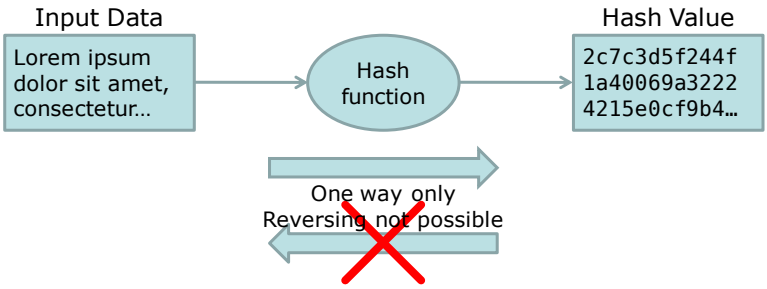
4.1. Hash Functions

A hash function is a computational procedure that takes an arbitrary block of data and returns a fixed-length summary, called a hash value or a digest of the original data block. Numerous hash functions have been invented over the years. The ones supported by the GuardTime timestamping service are listed in appendix "GuardTime Technical Reference – Formats and Algorithms".

Cryptographic hash functions are designed so that:

1. It is quite easy to compute the hash value of any given data block.
2. It is extremely difficult to construct a data block that has a given hash value (this property is called preimage resistance).
3. It is extremely difficult to modify a given data block so that its hash value does not change or construct a second data block with the same hash value as the given one (this property is called second preimage resistance).
4. It is extremely unlikely that two different data blocks will have the same hash value (this property is called collision resistance).

Figure 3: A hash function.



The expressions "extremely difficult" above mean there is no known way for this computation to be performed for any of the hash functions supported by GuardTime even if all the computers currently in the world would have worked only on this task for the lifetime of the Universe (even neglecting the fact that neither the computers nor the hash algorithms existed that long ago).

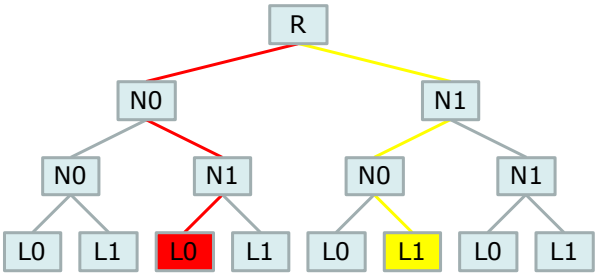
Due to these properties, hash values are often used in cryptography to represent the original data that might be too long or too confidential to handle directly. If the hash values computed from two data blocks match, it is assumed that the actual data blocks themselves must also match. Similarly, if a hash value computed from a data block matches the value computed earlier from the same block, it is assumed the data has not changed in the meantime.

4.2. Binary Trees

A binary tree is a hierarchy in which each parent node can have at most two child nodes (Figure 4). A node that is not a child of any other ("R" in the figure) is called the root of the tree, and a node that has no children ("L0" and "L1") is called a leaf. There is only one root node in a tree, but many leaf nodes.

In a binary tree, each child node is designated as a left ("N0" or "L0") or right ("N1" or "L1") child of its parent node. The path from the root to any leaf node is unique. For example, the only way to get from the root to the red leaf is to follow the red edges, that is, to go left, then right, and then left again. Conversely, the final destination of any sequence of "left" and "right" orders is also unique. For example, the sequence "right", "left", "right" leads from the root to the yellow leaf along the yellow edges.

Figure 4: A binary tree.



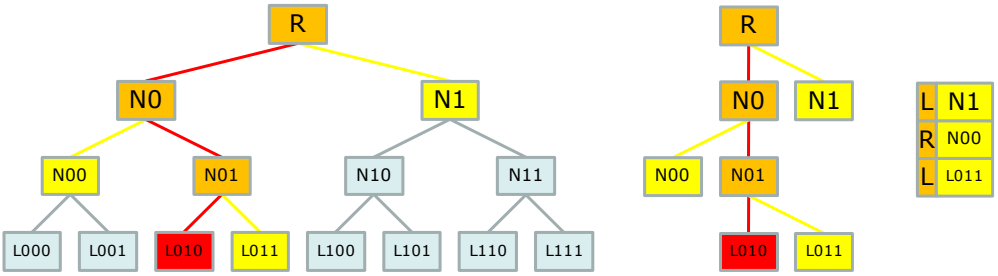
4.3. Merkle Trees

A Merkle tree is a binary tree where leaf nodes are filled with hash values of some input data and each non-leaf node is computed as the hash of the concatenation of the values in its two child nodes. The hashing step after the concatenation ensures the size of the values does not grow as we move from the leaf nodes towards the root of the tree.

After the tree is filled and the resulting value in the root node stored somewhere where its integrity is ensured, it is possible to prove the participation of any particular leaf node in the computation that led to the given value in the root node without the need to involve the whole tree in each proof.

For example, to prove that the current value in the red leaf node "L010" (Figure 5, left) really participated in the computation and has not been changed since, we need to re-compute the orange nodes on the path from the red leaf to the root of the tree. To perform these steps, we only need the hash values from the yellow sibling nodes and the information in which order to concatenate the two values in each step.

Figure 5: A Merkle tree, a hash computation, and a hash chain.



Indeed, with this knowledge we can restore the computation (Figure 5, center): concatenating the value of "L010" in question and the saved value of "L011", we can

compute the value of the node "N01"; concatenating the saved "N00" and the computed "N01", we can compute the value of "N0"; concatenating the computed "N0" and the saved "N1", we can compute the value of the root "R". If this matches the root value from the original computation, we can conclude that all the input data (and in particular the value of "L010") is the same as it was when the tree was first filled and the root value saved.

A complete binary tree with N layers below the root node has 2^N leaves. Thus, up to 2^N input hash values can be combined into a Merkle tree with N-step hash chains connecting each leaf to the root. Then anybody who has the root hash value of the original tree can verify the integrity of any of the 2^N leaf hash values. The owner of any leaf hash has to supply just the N additional hash values and N direction indicator values (Figure 5, right) to construct the proof linking their hash value to the saved root hash.

There are two ways an adversary could possibly fake this process. One would be to invent the values of the yellow nodes that would lead to the expected value in the root node even if the value in the red node has been changed. If the hash function is preimage resistant, it is impossibly difficult to do this. Thus, the only other option for the adversary would be to replace the saved root hash value. GuardTime prevents the second kind of attack by storing the root hash values of the aggregation trees in the calendar tree and periodically publishing the root hash value of the calendar tree in widely circulated newspapers.

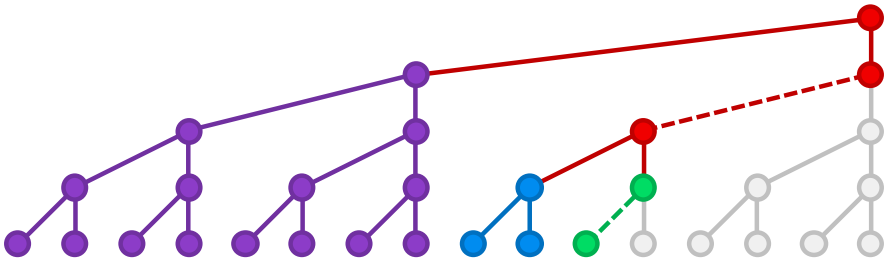
4.4. GuardTime Calendar

The GuardTime calendar is a special kind of Merkle tree. At any given moment, the tree contains a leaf node for each second since 1970-01-01 00:00:00 UTC. The leaves are numbered left to right starting from zero and new leaves are always added to the right.

There are two ways to describe how the tree is built and hash chains for individual seconds extracted from it.

One way is to imagine that the tree is built out to the maximal possible size at once and the leaf nodes corresponding to the future are empty (gray in Figure 6, Figure 7, Figure 8) and, instead of the regular Merkle concatenation and hashing, when two empty nodes are merged, the result is another empty node, and when a non-empty node is merged with an empty one, the non-empty value is just copied to the next level (dashed edges in the figures).

Figure 6: Sparse GuardTime calendar tree with $11_{10} = 1011_2$ leaves filled.



When hash chains are extracted from such a tree, the "copy" operations (the dashed edges in the figures) are not included in the hash chains. For example, the hash chain from the only green leaf in Figure 6 would consist of just two steps: the copying of the original hash value is skipped, the merging with the root of the blue sub-tree is included,

the copying of the resulting hash value is skipped, and merging with the root of the purple sub-tree is included.

Figure 7: Sparse GuardTime calendar tree with $12_{10} = 1100_2$ leaves filled.

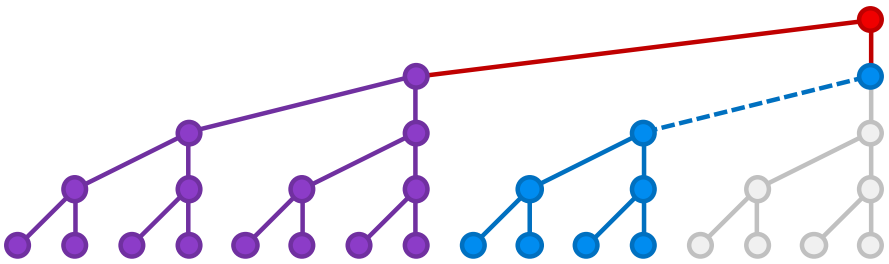
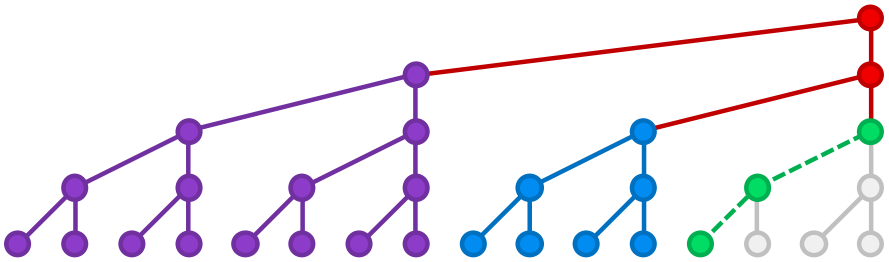


Figure 8: Sparse GuardTime calendar tree with $13_{10} = 1101_2$ leaves filled.

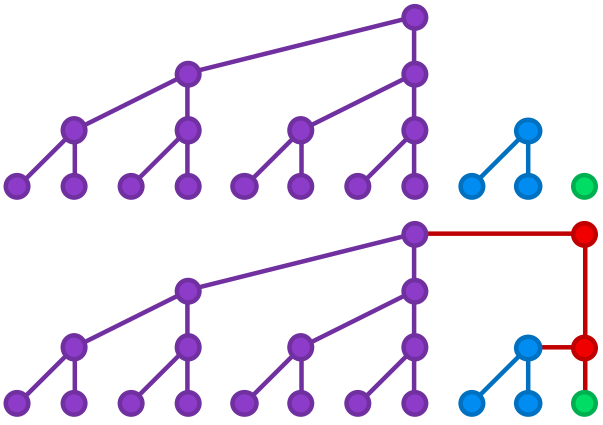


Another way to arrive at the same hash chains is to imagine that the calendar tree is built in two phases.

In the first phase, the leaves are collected into complete binary trees, starting from left, and making each tree as large as possible (Figure 9, Figure 10, Figure 11, top). Note that the sizes of the trees correspond to the 1-bits in the binary representation of the total number of leaves (each tree is drawn in the same color as the corresponding 1-bit in the caption). Such a collection of trees is called a forest.

In the second phase, the forest is turned into a single tree by merging the roots of the initial trees, but this time starting from the right (Figure 9, Figure 10, Figure 11, bottom) adding new parent nodes as needed (red nodes in the figures).

Figure 9: Compact GuardTime calendar tree with $11_{10} = 1011_2$ leaves.



Since this "compact" tree does not contain any special "copy" operations, the hash chains are extracted as from any Merkle tree. For example, the hash chain from the only green leaf in Figure 9 would consist of two steps: merging the input hash value with the root of the blue sub-tree and merging the result with the root of the purple sub-tree. As one would expect, the end result is the same as for the construction explained in Figure 6.

Figure 10: Compact GuardTime calendar tree with $12_{10} = 1100_2$ leaves.

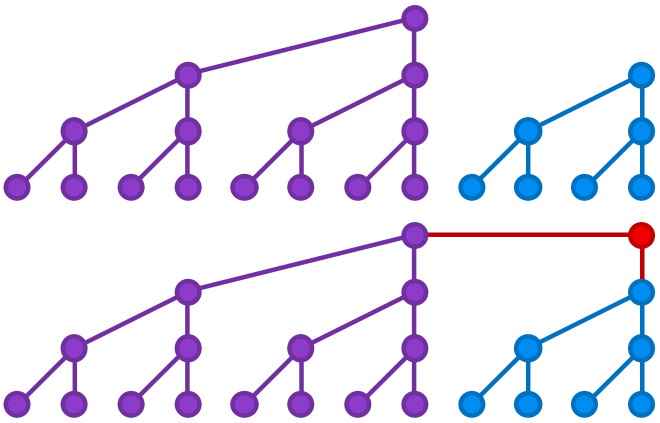
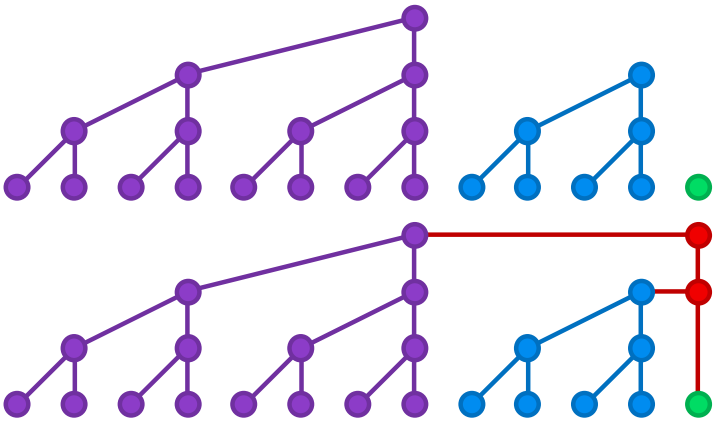


Figure 11: Compact GuardTime calendar tree with $13_{10} = 1101_2$ leaves.



Since the calendar tree is built in a deterministic manner, the shape of the tree for any moment can be reconstructed knowing just the number of leaf nodes in the tree at that moment, which is one more than the number of seconds from 1970-01-01 00:00:00 UTC to that moment.

Therefore, given the time when the calendar tree was created and a hash chain extracted from it, we can compute the time value corresponding to the leaf node belonging to the hash chain.

The algorithm for doing this relies on two facts regarding the compact calendar tree:

a) The left sub-tree of a node is always a complete binary tree.

b) The right sub-tree of a node has the same structure as the full calendar tree with the same number of leaves (which may also be a complete binary tree).

These two properties follow immediately from the way the calendar tree is constructed.

These two observations yield the following recursive algorithm:

```
// input:
//    r - time when the tree was created, in seconds from 1970-01-01 00:00:00 UTC
//    s - sequence of LEFT, RIGHT directions from root to the leaf
// output:
//    the time corresponding to the leaf, in seconds from 1970-01-01 00:00:00 UTC
function get_time(r, s)
   if empty(s) then
      assert(r = 0)
      return 0
   else
      assert(r > 0)
      h = hibit(r) // extract the value of highest 1-bit in r
      if head(s) = LEFT then
         // leaf is in the left subtree
         return get_time(h - 1, tail(s))
      else
         // leaf is in the right subtree
         return h + get_time(r - h, tail(s)
      end if
   end if
end function
```

The same algorithm can also be given non-recursively:

```
// input:
//    r - time when the tree was created, in seconds from 1970-01-01 00:00:00 UTC
//    s - sequence of LEFT, RIGHT directions from root to the leaf
// output:
//    the time corresponding to the leaf, in seconds from 1970-01-01 00:00:00 UTC
function get_time(r, s)
   t = 0 // accumulator for result
   for i = 1 to length(s)
      assert(r > 0)
      h = hibit(r) // extract the value of highest 1-bit in r
      if s[i] = LEFT then
         // leaf is in the left subtree
         r = h - 1
      else
         // leaf is in the right subtree
         t = t + h
         r = r - h
      end if
   end for
   assert(r = 0)
   return t
end function
```

5. Timestamping Process

GuardTime timestamps follow the structure defined by the RFC 3161 timestamping standard. The standard defines an internal hierarchical structure for the timestamps and requires certain sections of the structure to be linked to each other using hashes and a final hash value to be signed by the timestamping authority.

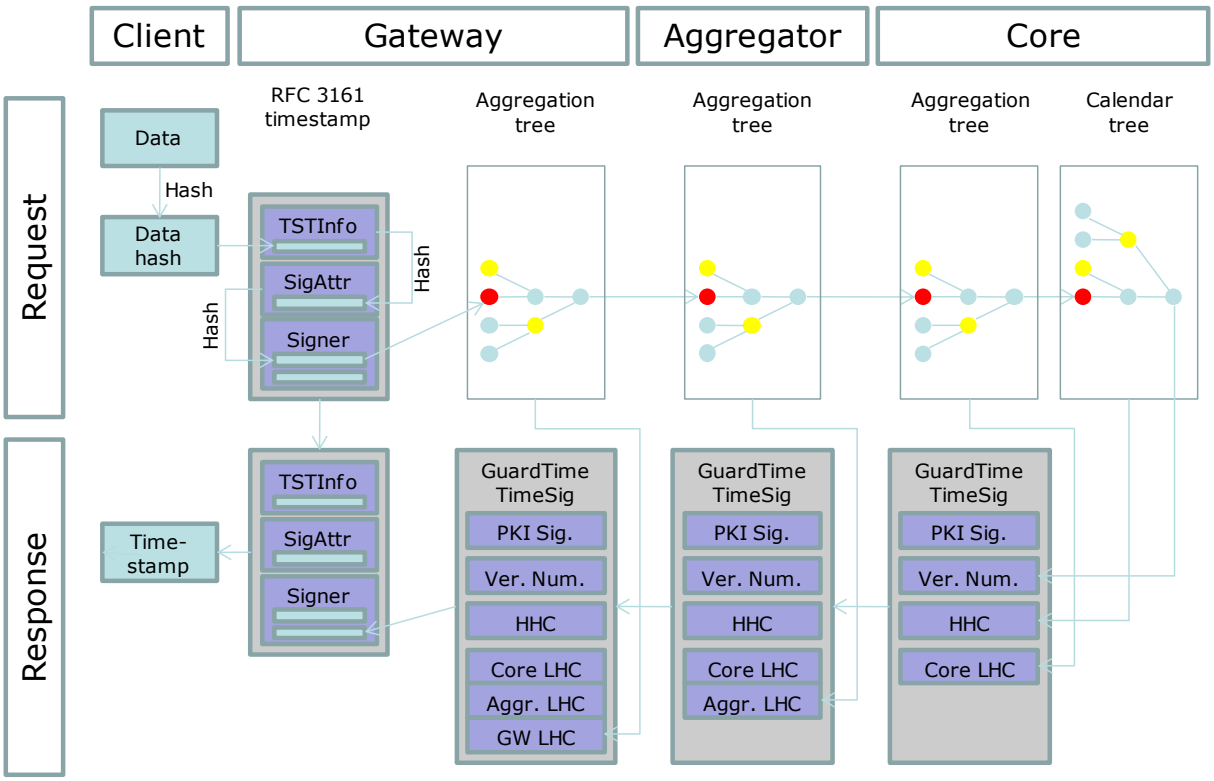
The main distinction of GuardTime timestamps is the use of the GuardTime TimeSignature algorithm for signing the final hash value. The GuardTime TimeSignature, unlike the more common PKI-based digital signatures, is assembled in a distributed grid and does not depend on a secret key for long-term verification.

The aim of this section is to explain how the signature is assembled and the information needed for verification is published.

5.1. Aggregation and Distribution

For a more detailed understanding of the timestamping process, consider Figure 12.

Figure 12: Aggregation and TimeSignature assembly.



The process is initiated by a client that hashes the data to be timestamped. Next the client passes the hash value to the GuardTime gateway that creates a timestamp template for the request and starts the signing process (top left in the figure).

As already explained, the GuardTime aggregation grid is organized in layers. Each layer operates in discrete rounds. Each aggregator collects the requests from the clients for one round and builds a temporary Merkle tree of the hash values to be signed (top center in the figure). The hash value in the root of the tree is passed on to the next aggregation layer that builds another temporary Merkle tree, and passes the root hash value on to the next layer. The whole set of aggregation trees can be viewed as one giant distributed

Merkle tree if we imagine the hash values in the leaves of upper level trees to be replaced with the lower level trees they represent.

This process continues up the aggregation hierarchy until the core cluster inserts the root hash value from the global aggregation tree into the persistent calendar tree (top right in the figure).

Now the aggregation process is complete and assembly and distribution of the signature starts (bottom right in the figure). First, the current time and the root hash value of the calendar tree (together known as the Verification Number) are temporarily signed by a PKI-based signature. This is needed to authenticate the signature until the next Integrity Code is published (as explained in the following section).

Then the hash chain leading from the root of the calendar tree to the leaf for the current second is extracted and stored in the TimeSignature structure as the history hash chain (HHC in the figure).

Then, for each subordinate aggregator, the hash chain leading from the root of the aggregation tree to the leaf corresponding to that aggregator is extracted and added to the TimeSignature as location hash chain (Core LHC in the figure), and the results sent to the corresponding subordinate aggregators. When this is done, the core aggregation tree is discarded, as it is not needed for the next round.

Going down the hierarchy, each aggregation layer receives a partial TimeSignature from the level above, makes as many copies of it as there are subordinates on the next lower level, appends the hash chain from the root of its aggregation tree to the corresponding leaf to each one (Aggr. LHC in the figure), sends the updated TimeSignatures on to the next level, and then discards the aggregation tree.

Finally, the gateway completes the TimeSignatures by adding the last slice of the location hash chain (GW LHC in the figure) to each one and inserts the signatures into the timestamps. It then returns the complete timestamps to the clients (bottom left in the figure) and discards its own aggregation tree to clean up the resources for the next round.

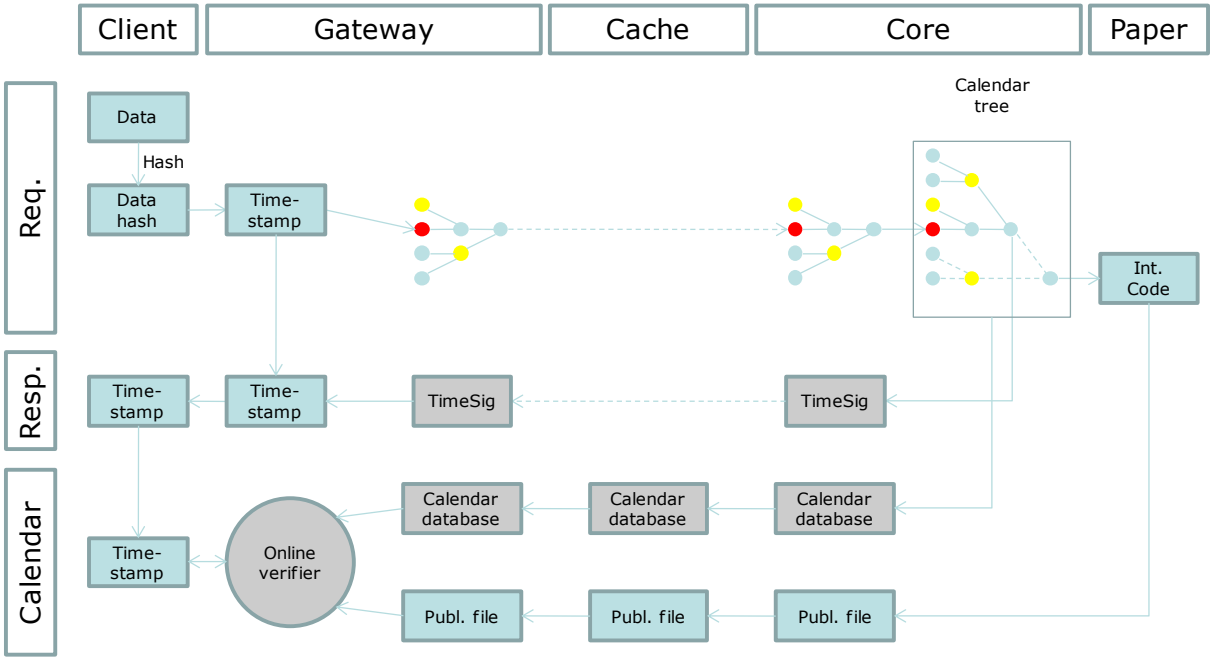
5.2. Publishing

Periodically, the current state of the calendar is published in printed newspapers as the Integrity Code (Figure 13). Any timestamp issued prior to the creation of the Integrity Code can be linked to it, thus proving the age and integrity of the signed data.

A timestamp originally contains only the hash chain leading from the corresponding leaf of the calendar tree up to the root for the moment when the timestamp was created. The parts of the calendar tree shown in dashed lines did not yet exist when the timestamp was created. Thus, the sibling hash values from this part of the tree were not yet known and couldn’t be included in the timestamp.

To supply the missing data for verification, the full calendar tree is distributed back to the gateway level. Also, a publications file containing the full list of Integrity Codes is distributed to enable the online verification service (described in more detail in the following section).

Figure 13: Publishing.



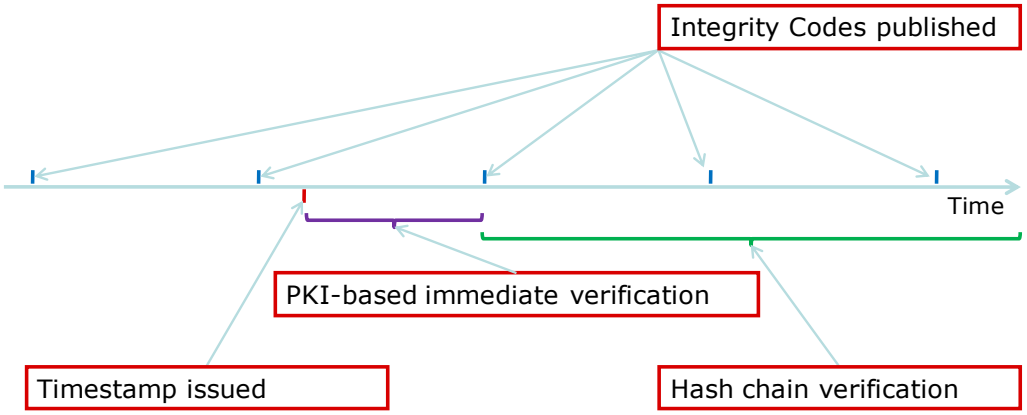
6. Verification Process

The previous section described how the GuardTime timestamp are assembled and the Integrity Codes published. This section explains how the published information is afterwards used to verify the timestamps.

6.1. Timeline

To understand the context for the GuardTime verification algorithm, let's first consider the timeline shown in Figure 14. The Integrity Codes are published at regular intervals (blue ticks on the timeline). Suppose a timestamp is issued (the red tick). It will take some time (the purple bracket) until the publication of the first Integrity Code that would cover the timestamp.

Figure 14: Verification timeline.



Until an Integrity Code is available, the TimeSignature can be authenticated based on GuardTime's conventional digital signature included inside the TimeSignature. After an Integrity Code is published, the verification does not have to depend on the GuardTime signing keys and can instead be based on the public calendar data.

6.2. Trust Models

A cryptographic verification process could be likened to a mathematical proof.

In mathematics, theorems are proven by using logical inference steps to reduce the claims to basic truths called axioms. It could be said theorems in mathematics are not absolute, but relative to the axioms.

A verification process proves the authenticity of the data in question by relating them to something that is so hard to forge that it can intrinsically be relied on to be authentic – a trust anchor. Deciding on what to use as the trust anchors and which cryptographic primitives to use as the inference steps is called the trust model.

The long-term trust model of GuardTime timestamps relies only on hash functions as the inference steps and printed newspapers as the trust anchors.

For convenience and performance, authenticated electronic archives may be used instead of printed newspapers.

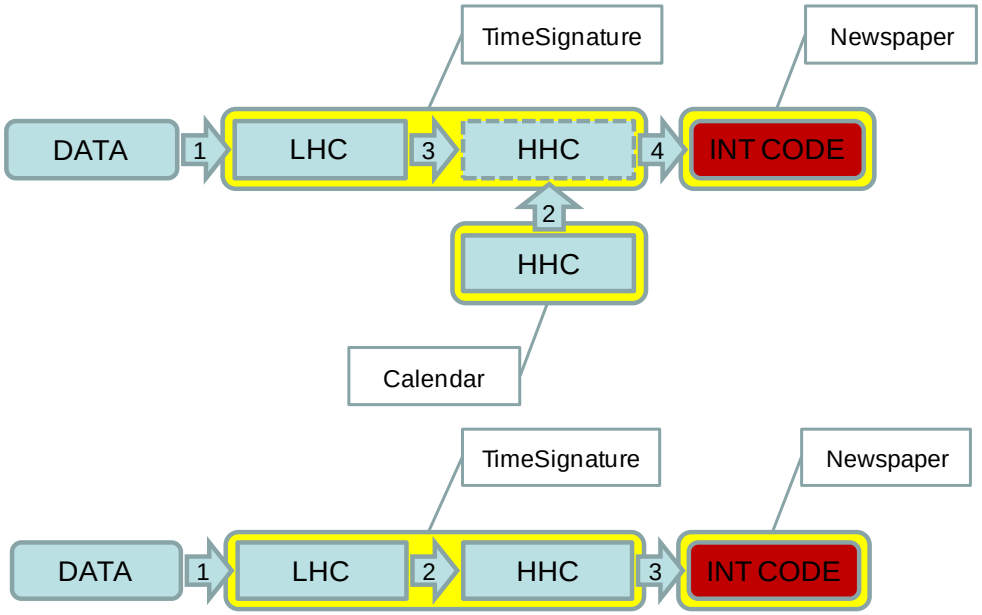
Conventional digital signatures are used for short-term authentication until a printed publication is available. It is not needed to rely on security of any signing keys for long-term verification.

6.2.1. Verification against Newspaper

For the strongest proof, a GuardTime TimeSignature can be verified using an Integrity Code published in a printed newspaper (Figure 15, top):

1. The timestamp data is protected by the location hash chain (LHC) in the TimeSignature.
2. A history hash chain (HHC) is extracted from the GuardTime calendar data. Normally the verification component of GuardTime gateway supplies this.
3. The LHC is protected by the HHC from the GuardTime calendar.
4. The HHC is protected by the Integrity Code in the newspaper.
5. The Integrity Code in printed newspaper is the trust anchor.

Figure 15: Verification of TimeSignature against newspaper publication.



It is possible to take the history hash chain mentioned in previous description and store it inside the TimeSignature. Then future verifications can be done without access to the gateway (Figure 15, bottom):

1. The timestamp data is protected by the location hash chain (LHC) in the TimeSignature.
2. The LHC is protected by the history hash chain (HHC) in the TimeSignature.
3. The HHC is protected by the Integrity Code in the newspaper.
4. The Integrity Code in printed newspaper is the trust anchor.

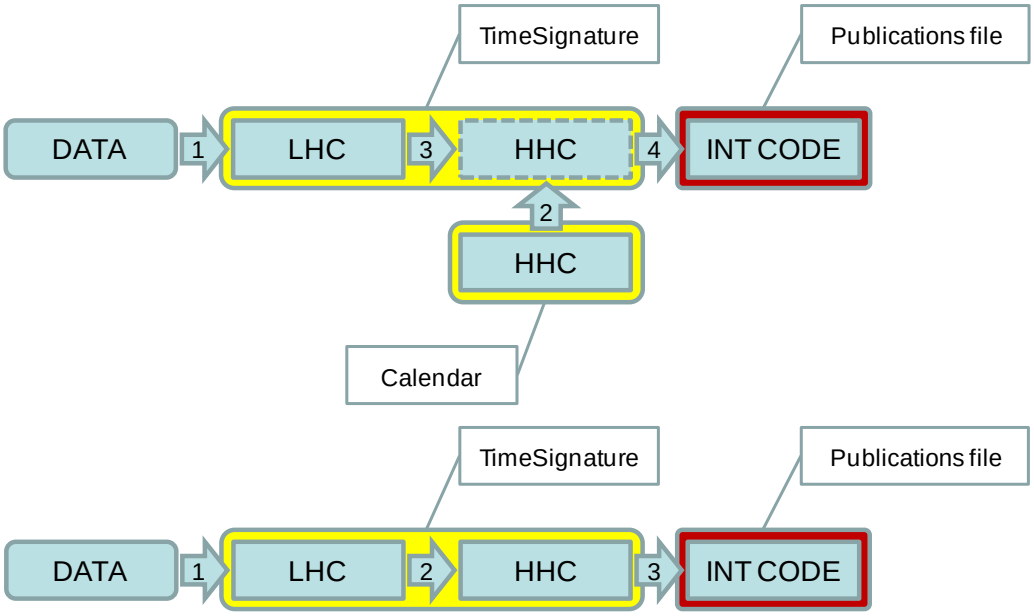
6.2.2. Verification against Publications File

For convenience and performance reasons, it is not always desirable to refer to a printed newspaper. Verification using an electronic publications file is very similar; the only

difference is that an authenticated electronic document is used instead of a printed one (Figure 16, top):

1. The timestamp data is protected by the location hash chain (LHC) in the TimeSignature.
2. A history hash chain (HHC) is extracted from the GuardTime calendar data. Normally the verification component of GuardTime gateway supplies this.
3. The LHC is protected by the HHC from the GuardTime calendar.
4. The HHC is protected by the Integrity Code in the publications file.
5. The publications file is the trust anchor. This file must be properly authenticated before it is used. This is discussed in Section 6.2.4.

Figure 16: Verification of TimeSignature against electronic publication.



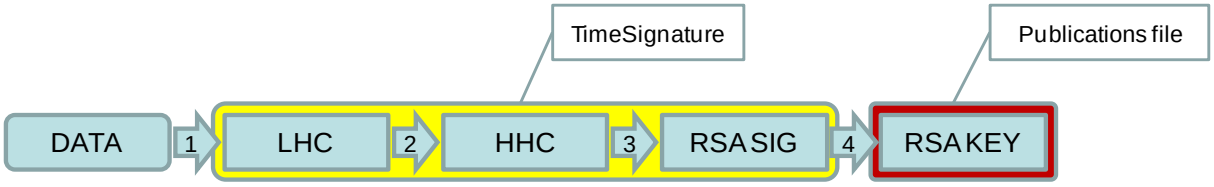
Like in the case with printed newspaper, it is possible to store the history hash chain inside the TimeSignature and perform future verifications without access to the gateway (Figure 16, bottom).

6.2.3. Verification of Recent TimeSignature

An Integrity Code protects all TimeSignatures issued before the code was generated. In short term (until the next Integrity Code is generated and published), a recently issued signature can be verified as follows (Figure 17):

1. The timestamp data is protected by the location hash chain (LHC) in the TimeSignature.
2. The LHC is protected by the (temporary) history hash chain (HHC) in the TimeSignature.
3. The HHC is protected by GuardTime's conventional digital signature embedded inside the TimeSignature.
4. GuardTime's signing keys are listed in the publications file.
5. The publications file is the trust anchor. This file must be properly authenticated before it is used. This is discussed in Section 6.2.4.

Figure 17: Verification of recently issued TimeSignature.

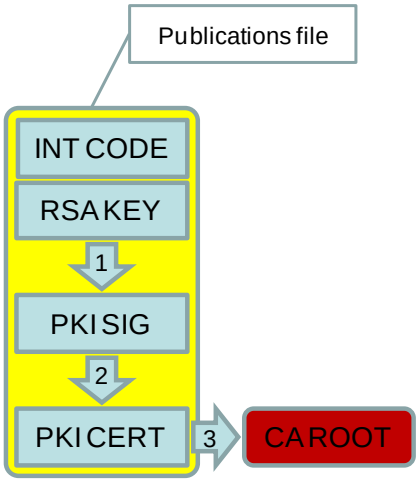


6.2.4. Authentication of Publications File

Just as a printed newspaper for verification has to be acquired from a reliable source, an electronic publications file has to be properly authenticated before it can be used as a trust anchor. This can be done semi-automatically using the PKI framework as follows (Figure 18):

1. The Integrity Codes and GuardTime's timestamp signing keys in the publications file are signed using GuardTime's file signing key.
2. GuardTime's file signing key is authenticated by a certificate issued by a well-known PKI certification authority (CA).
3. The CA's certificate signing key is authenticated by a CA root certificate.
4. The CA root certificate could be taken as a trust anchor implicitly or it could be verified by contacting the CA.

Figure 18: Authenticating GuardTime publications file.



It is quite conceivable that in a security critical environment any root certificate is verified offline before it is used as a trust anchor. Taking it a step further, a customer may also decide to verify the GuardTime publications file manually before allowing it to be used for timestamp verification process.

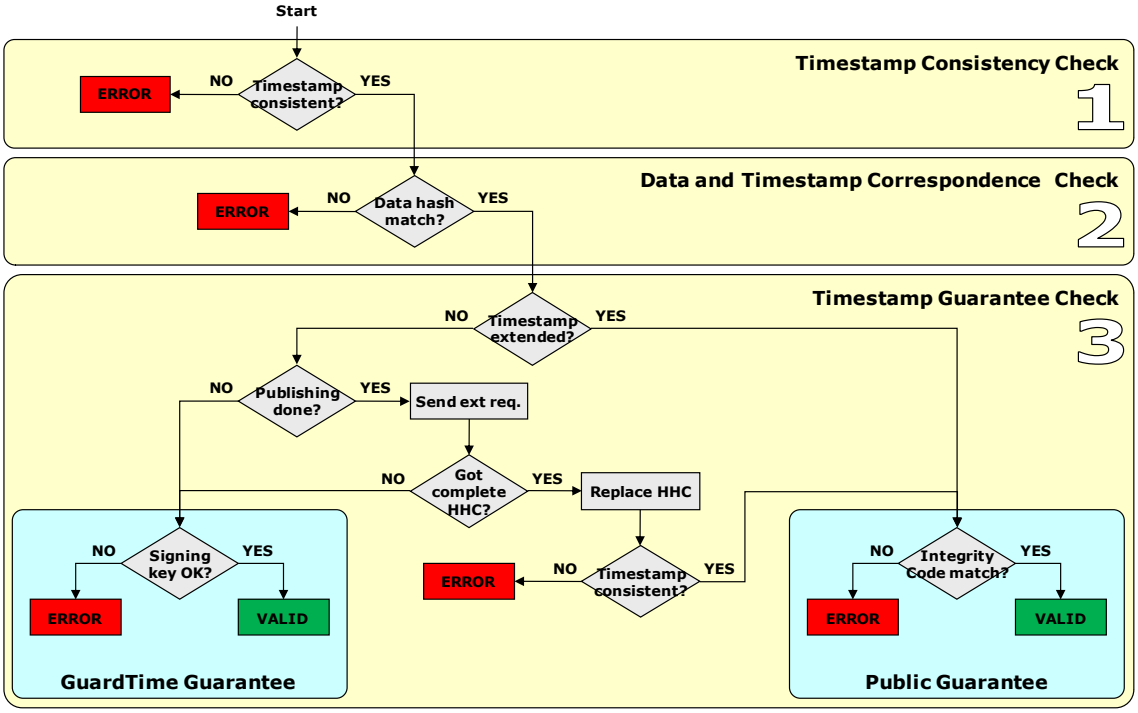
6.3. Verification Algorithm

The overall verification algorithm for GuardTime timestamps (also illustrated in Figure 19) is as follows:

1. Check that the timestamp is internally consistent (specified in more detail in Section 6.3.1). If it is not, terminate with the error message returned by the consistency check.

2. Check that the hash value computed form the data matches the one stored inside the timestamp. If it does not, terminate with the error message "Wrong Document Failure".
3. Check if the timestamp is already extended (that is, the history hash chain in the TimeSignature leads to an Integrity Code published in the newspaper). If it is, go to step 7.
4. Check if the timestamp is old enough to be extended. More specifically, if the timestamp is more recent than the newest Integrity Code in the publications file, go to step 8.
5. Check if a GuardTime gateway is available for online verification. If it is, send an extending request to the gateway. If there is no gateway, or it does not respond, or responds with an error message, go to step 8.
6. Extend the timestamp using the hash chain and the publication information returned by the gateway (specified in more detail in Section 6.3.2). If the extending fails, terminate with the error message returned by the extending procedure.
7. Check if the Verification Number in the timestamp matches an Integrity Code either in the publications file or a newspaper. If it does, terminate with the success message "Publication Checked". If it does not, terminate with the error message "Not Valid Publication".
8. Fallback to PKI-based verification: Check if the signing key used to sign the Verification Number is in the list of valid GuardTime timestamp signing keys in the publications file. If it is, terminate with the success message "PKI Signature Checked". If it is not, terminate with the error message "Not Valid Public Key".

Figure 19: Verification flowchart.



6.3.1. Timestamp Consistency Check

The consistency check of a timestamp is performed as follows:

1. Syntax check: verify that:
    - the timestamp is a valid ASN.1 message having the structure defined in the appendix "GuardTime Technical Reference – Formats and Algorithms";
    - the timestamp does not contain any unknown extensions marked as critical;
    - that the correct content type and message digest are among the signed attributes (see the Appendix 6.5.3 for more details).

    If any of these checks fail, return the error message "Syntactic Check Failure".
2. Hash chain check: check that the hash chain computation (defined in the appendix "GuardTime Technical Reference – Formats and Algorithms") yields the value recorded in the Verification Number. If it does not, return the error message "Hash Chain Verification Failure".
3. PKI signature check: if there is a PKI signature inside the TimeSignature, check that the PKI signature is a properly formed PKCS#1 digital signature on the Verification Number. If it is not, return the error message "PKI Signature Failure".

6.3.2. Timestamp Extending

The purpose of extending a timestamp is to provide a connection from the timestamp to an Integrity Code published in physical media. The process to do this is as follows:

1. Extract the time value corresponding to the leaf on the calendar tree to which the timestamp is connected (see Section 4.4).
2. Send a request for a hash chain connecting the leaf to a published Integrity Code. This is done using the protocol given in the appendix "GuardTime Technical Reference – Formats and Algorithms". If there is no extending service available, or the extending service does not have the calendar data needed to return the hash chain, the timestamp can't be extended and the verification will have to fall back to PKI-based signature.
3. Check that the response returned from the extending service agrees with the timestamp: verify that:
    - the history hash chain in the response refers to the same time value as the history hash chain in the timestamp;
    - the hash values coming from the past relative to the timestamp are the same in the history hash chain in the response as in the history hash chain in the timestamp.

    If either of these checks fails, return the error message "Cannot Extend".
4. Replace the history hash chain and the publication data in the timestamp with the ones returned by the extending service. Discard the PKI signature and the certificate for the PKI signing key from the timestamp.

7. Evidence Package

In this section, we examine the package that needs to be presented to a third party in order to prove the integrity of some data timestamped using GuardTime technology.

If the timestamp in question is old enough to be extended (which we assume will be the case most of time and almost always for a trial in a court of law), it is recommended that an extended timestamp (that is, a timestamp where the history hash chain extracted from the calendar data has been inserted into the timestamp) will be used.

In this case, the following is sufficient to independently verify the timestamp:

1. The original data that was timestamped.
2. The extended timestamp.
3. A verification tool that can be either one obtained from GuardTime or one developed independently based on specifications from GuardTime. Presumably the court will have an expert witness review the tool to testify that it indeed properly implements the verification algorithm.
4. A published Integrity Code to check the Verification Number computed by the verification tool. The Integrity Code could be delivered either as part of the GuardTime publications file or as a printed in a newspaper; it is expected that a printed newspaper will be considered stronger form of proof.

In a rare case where the proof is required before an Integrity Code has been published, the following is sufficient to verify the timestamp based on GuardTime's digital signature:

1. The original data that was timestamped.
2. The PKI-signed timestamp.
3. A verification tool that can be either one obtained from GuardTime or one developed independently based on specifications from GuardTime. Presumably the court will have an expert witness review the tool to testify that it indeed properly implements the verification algorithm.
4. A list of GuardTime signing keys in the form of a GuardTime publications file.

8. Format Specifications

Detailed data format specifications are given in a separate document "GuardTime Technical Reference – Formats and Algorithms".