DTKI: a new formalised PKI with no trusted parties

JIANGSHAN YU, School of Computer Science, University of Birmingham, UK
VINCENT CHEVAL, LORIA, CNRS, France
MARK RYAN, School of Computer Science, University of Birmingham, UK

The security of public key validation protocols for web-based applications has recently attracted attention because of weaknesses in the certificate authority model, and consequent attacks.

Recent proposals using public logs have succeeded in making certificate management more transparent and verifiable. However, those proposals involve a fixed set of authorities. This means an oligopoly is created. Another problem with current log-based system is their heavy reliance on trusted parties that monitor the logs.

We propose a distributed transparent key infrastructure (DTKI), which greatly reduces the oligopoly of service providers and removes the reliance on trusted parties. In order to be able to precisely state and prove properties of DTKI, we formalise a new public log construction which is employed in DTKI to provide full transparency. In addition, we provide a formal analysis of the security that DTKI guarantees.

Categories and Subject Descriptors: []

General Terms: Security, Verification

Additional Key Words and Phrases: Authenticity, certificate, Public key infrastructure, Public log, TLS

ACM Reference Format:
ACM Trans. Info. Syst. Sec. V, N, Article A (YYYY), 70 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The security of web-based applications such as e-commerce and web-mail depends on the ability of a user's browser to obtain authentic copies of the public keys for the application website. For example, suppose a user wishes to log in to her bank account through her web browser. The web session will be secured by the public key of the bank. If the user's web browser accepts an inauthentic public key for the bank, then the traffic (including log-in credentials) can be intercepted and manipulated by an attacker.

The authenticity of keys is assured at present by *certificate authorities* (CAs). In the given example, the browser is presented with a public key certificate for the bank, which is intended to be unforgeable evidence that the given public key is the correct one for the bank. The certificate is digitally signed by a CA. The user's browser is pre-configured to accept certificates from certain known CAs. A typical installation of Firefox has about 100 root certificates in its database.

Unfortunately, numerous problems with the current CA model have been identified. Firstly, CAs must be assumed to be trustworthy. If a CA is dishonest or compromised, it may issue certificates asserting the authenticity of fake keys; those keys could be created by an attacker or by the CA itself. Secondly, the assumption of honesty does not scale up very well. As already mentioned, a browser typically has hundreds of CAs registered in it, and the user cannot be expected to have evaluated the trustworthiness and security of all of them. This fact has been exploited by attackers [Eckersley 2011; Leyden 2012; cer ; Roberts 2011; Sterling 2011; Falliere et al. 2011]. In 2011, two CAs were compromised: Comodo [Appelbaum 2011] and DigiNotar [cer 2012]. In both cases, certificates for high-profile sites were illegitimately obtained, and in the second case, reportedly used in a *man in the middle* (MITM) attack [Arthur 2011]. See [Niemann and Brendel 2013] for a survey on CA compromises.

Proposed solutions

Several interesting solutions have been proposed to address these problems. For a comprehensive survey, see [Clark and van Oorschot 2013].

Key pinning mitigates the problem of untrustworthy CAs, by defining in the client browser the parameters concerning the set of CAs that are considered entitled to certify the key for a given domain [Langley 2011; Marlinspike and Perrin 2012]. However, scalability is a challenge for key pinning.

Crowd-sourcing techniques have been proposed in order to detect untrustworthy CAs, by enabling a browser to obtain warnings if the received certificates are different from those that other people are being offered [Wendlandt et al. 2008; Eckersley and Burns ; Alicherry and Keromytis 2009; Amann et al. 2012; obs ; ; Soghoian and Stamm 2011; Marlinspike 2011]. Crowd-sourcing techniques have solved some CA-based problems. However, the technique cannot distinguish between attacks and authentic certificate updates, and may also suffer from an initial unavailability period.

Solutions for revocation management of certificates have also been proposed; they mostly involve periodically pushing revocation lists to browsers, in order to remove the need for on-the-fly revocation checking [Rivest 1998; Langley 2012]. However, these solutions create a window during which the browser's revocation lists are out of date until the next push.

More recently, solutions involving public append-only logs have been proposed. We consider the leading proposals here.

Public log-based systems. *Sovereign Keys* (SK) [Eckersley 2012] aims to get rid of browser certificate warnings, by allowing domain owners to establish a long term ("sovereign") key and by providing a mechanism by which a browser can hard-fail if it doesn't succeed in establishing security via that key. The sovereign key is used to cross-sign operational TLS [Dierks and Rescorla 2008; Turner and Polk 2011] keys, and it is stored in an append-only log on a "time-line server", which is abundantly mirrored. However, in SK, internet users and domain owners have to trust mirrors of time-line servers, as SK does not enable mirrors to provide efficient verifiable proofs that the received certificate is indeed included in the append-only log.

Certificate transparency (CT) [Laurie et al. 2013] is a technique proposed by Google that aims to efficiently detect fake public key certificates issued by corrupted certificate authorities, by making certificate issuance transparent. They improved the idea of SK by using append-only Merkle tree to organise the append-only log. This enables the log maintainer to provide two types of verifiable cryptographic proofs: (a) a proof that the log contains a given certificate, and (b) a proof that a snapshot of the log is an extension of another snapshot (*i.e.*, only appends have taken place between the two snapshot). The time and size for proof generation and verification are logarithmic in the number of certificates recorded in the log. Domain owners can obtain the proof that their certificates are recorded in the log, and provide the proof together with the certificate to their clients, so the clients can get a guarantee that the received certificate is recorded in the log.

Accountable key infrastructure (AKI) [Kim et al. 2013] also uses public logs to make certificate management more transparent. By using a data structure that is based on lexicographic ordering rather than chronological ordering, they solve the problem of key revocations in the log. In addition, AKI uses the "checks-and-balances" idea that allows parties to monitor each other's misbehaviour. So AKI limits the requirement to trust any party. Moreover, AKI prevents attacks that use fake certificates rather than merely detecting such attacks (as in CT). However, as a result, AKI needs a strong assumption — namely, CAs, public log maintainers, and validators do not collude together — and heavily relies on third parties called validators to ensure that the log is maintained without improper modifications.

Certificate issuance and revocation transparency (CIRT) [Ryan 2014] is a proposal for managing certificates for end-to-end encrypted email. It proposes an idea to address the revocation problem left open by CT, and the trusted party problem of AKI. It collects ideas from both CT and AKI to provide transparent key revocation, and reduces reliance on trusted parties by designing the monitoring role so that it can be distributed among user browsers. However, CIRT can only detect attacks that use fake certificates; it cannot prevent them. In addition, since CIRT was proposed for email applications, it does not support the multiplicity of log maintainers that would be required for web certificates.

Attack Resilient Public-Key Infrastructure (ARPKI) [Basin et al. 2014] is an improvement on AKI. In ARPKI, a client can designate n service providers (e.g. CAs and log maintainers), and only needs to contact one CA to register her certificate. Each of the designated service providers will monitor the behaviour of other designated service providers. As a result, ARPKI prevents attacks even when $n-1$ service providers are colluding together, whereas in AKI, an adversary who successfully compromises two out of three designated service providers can successfully launch attacks [Basin et al. 2014]. In addition, the security property of ARPKI is proved by using a protocol verification tool called Tamarin prover [Meier et al. 2013]. The weakness of ARPKI is that all n designated service providers have to be involved in all the processes (i.e. certificate registration, confirmation, and update), which would cause considerable extra latencies and the delay of client connections.

In public log-based systems, efforts have been made to integrate *revocation management* with the certificate auditing. CT introduced revocation transparency (RT) [Laurie and Kasper 2012a] to deal with certificate revocation management; and in AKI and ARPKI, the public log only stores currently valid certificates (revoked certificates are purged from the log). However, the revocation checking processes in both RT and A(RP)KI are linear in the number of issued certificates making it inefficient. CIRT allows efficient proofs of non-revocation, but it does not scale to multiple logs which are required for web certificates.

Remaining problems

A foundational issue is the problem of *oligopoly.* The present-day certificate authority model requires that the set of global certificate authorities is fixed and known to every browser, which implies an oligopoly. Currently, the majority of CAs in browsers are organisations based in the USA, and it is hard to become a browser-accepted CA because of the strong trust assumption that it implies. This means that a Russian bank operating in Russia and serving Russian citizens living in Russia has to use an American CA for their public key. This cannot be considered satisfactory in the presence of mutual distrust between nations regarding cybersecurity and citizen surveillance, and also trade sanctions which may prevent the USA offering services (such as CA services) to certain other countries.

None of the previously discussed public log-based systems address this issue. In each of those solutions, the set of log maintainers (and where applicable, time-line servers, validators, etc.) is assumed to be known by the browsers, and this puts a high threshold on the requirements to become a log maintainer (or validator, etc.). Moreover, none of them solve the problem that a multiplicity of log maintainers reduces the usefulness of transparency, since a domain owner has to check each log maintainer to see if it has mis-issued certificates. This can't work if there is a large number of log maintainers operating in different geographical regions, each one of which has to be checked by every domain owner.

A second issue is the requirement of trusted parties. Currently, all existing proposals have to rely on some sort of trusted parties or at least assume that not all parties are colluding together. However, a strong adversary (e.g. a government agency) might be able to control all service providers (used by a given client) in a system.

A third foundational issue of a different nature is that of analysis and correctness. SK, CT, AKI and CIRT are large and complex protocols involving sophisticated data structures, but none of them have been subjected to rigorous analysis. It is well-known that security protocols are notoriously difficult to get right, and the only way to avoid this is with systematic verification. For example, attacks on AKI and CIRT have been identified in [Basin et al. 2014] and in our appendix A, respectively. The flaws may be easily fixed, but only once they have been identified. It is therefore imperative to verify this kind of complex protocol. ARPKI is the first formally verified log-based PKI system. However, they used several abstractions during modelling in Tamarin prover. For example, they represent the underlying log structure (a Merkle tree) as a list. However, in systems like CIRT and this papr with more complex data structures, it is important to have a formalised data structure and its properties to prove the security claim. The formalisation of complex data structures and their properties in the log-based PKI systems is a remaining problem.

The last problem is the management of certificate revocation. As explained previously, existing solutions for managing certificate revocation (e.g. CRL, OCSP, RT) are still unsatisfactory.

This paper

We propose a new public log-based architecture for managing web certificates, called *Distributed Transparent Key Infrastructure* (DTKI), with the following contributions.

— We identify *anti-oligopoly* as an important property for web certificate management which has hitherto not received attention.
— Compared to its predecessors, DTKI is the first system to have all desired features — it minimises the presence of oligopoly, prevents attacks that use fake certificates, provides a way to manage certificate revocation, does not rely on any trusted party, and is secure even if all service providers (e.g. CAs and log maintainers) collude together (see Section 5 for our security statement). A comparison of the properties of different log-based systems can be found in Section 6.
— We provide formal machine-checked verification of its core security property using the Tamarin prover. In addition, we formalise the data structures needed for transparent public logs, and provide rigorous proofs of their properties.

2. OVERVIEW OF DTKI

Distributed Transparent Key Infrastructure (DTKI) is an infrastructure for managing keys and certificates on the web in a way which is *transparent*, minimises *oligopoly*, and eliminates the need for trusted parties. In DTKI, we mainly have the following agents:

Certificate log maintainers (CLM): A CLM maintains a database of all valid and invalid (e.g. expired or revoked) certificates for a particular set of domains for which it is responsible. It commits to digests of its log, and provides efficient proofs of presence and absence of certificates in the log with respect to the digest. CLMs behave transparently: their actions can be verified and therefore they do not require to be trusted.

A mapping log maintainer (MLM): To minimise oligopoly, DTKI does not fix the set of certificate logs. The MLM maintains association between certificate logs and the domains they are responsible for. It also commits to digests of the log, and provides efficient proof of current association, and behaves transparently. Clients of MLM are not required to trust the MLM, because they can efficiently verify the obtained associations.

Mirrors: Mirrors are servers that maintain a full copy of the mapping log and certificate logs respectively downloaded from the MLM and corresponding CLMs, and the corresponding digest of the log signed by the log maintainer. In other words, mirrors are distributed copies of logs. Anyone (e.g. ISPs, CLMs, CAs, domain owners) can be a mirror. Unlike in

SK, mirrors are not required to be trusted in DTKI, because they give a proof for every association that they send to their clients. The proof is associated to the digest of the MLM.

Certificate authorities (CA): They check the identity of domain owners, and create certificates for the domain owners' keys. However, in contrast with today's CAs, the ability of CAs in DTKI is limited since the issuance of a certificate from a CA is not enough to convince web browsers to accept the certificate (proof of presence in the relevant CLM is also needed).

In DTKI, each domain owner has two types of certificate, namely TLS certificate and master certificate. Domain owners can have different TLS certificates but can only have one master certificate. A TLS certificate contains the public key of a domain server for a TLS connection, whereas the master certificate contains a public key, called "master verification key". The corresponding secret key of the master certificate is called "master signing key". Similar to the "sovereign key" in SK [Eckersley 2012], the master signing key is only used to validate a TLS certificate (of the same subject) by issuing a signature on it. This limits the ability of certificate authorities since without having a valid signature (issued by using the master signing key), the TLS certificate will not be accepted. Hence, the TLS secret key is the one for daily use; and the master signing key is rarely used. It will only be used for validating a new certificate, or revoke an existing certificate. We assume that domain owners can take care of their master signing key.

After a domain owner obtains a master certificate or a TLS certificate from a CA, he needs to make a registration request to the corresponding CLM to publish the certificate into the log. To do so, the domain owner signs the certificate using the master signing key, and submits the signed certificate to a CLM determined (typically based on the top-level domain) by the MLM. The CLM checks the signature, and accepts the certificate by adding it to the certificate log if the signature is valid. The process of revoking a certificate is handled similarly to the process of registering a certificate in the log.

When establishing a secure connection with a domain server, the browser receives a corresponding certificate and proofs from a mirror of the MLM and a CLM, and verifies the certificate, the proof that the certificate is valid and recorded in the certificate log, and proof that this certificate log is authorised to manage certificates for the domain. Users and their browsers only accept a certificate if the certificate is issued by a CA, and validated by the domain owner, and current in the certificate log.

Fake master certificates or TLS certificates can be easily detected by the domain owner, because the CA will have had to insert such fake certificates into the log (in order to be accepted by browsers), and is thus visible to the domain owner.

Rather than relying on trusted parties (e.g. monitors in CT and validators in AKI) to verify the healthiness of logs and the relations between logs, DTKI uses a crowdsourcing-like way to ensure the integrity of the log and the relations between mapping log and a certificate log, and between certificate logs. In particular, the monitoring work in DTKI can be broken into independent little pieces, and thus can be done by distributing the pieces to users' browsers. In this way, users' browsers can perform randomly-chosen pieces of the monitoring role in the background (e.g. once a day). Thus, web users can collectively monitor the integrity of the logs.

To avoid the case that attackers create a "bubble" (i.e. an isolated environment) around a victim, we share the same assumption as other existing protocols (e.g. CT and CIRT) – we assume that gossip protocols [Jelasity et al. 2007] are used to disseminate digests of the log. So, users of logs can detect if a log maintainer shows different versions of the log to different sets of users. Since log maintainers sign and time-stamp their digests, a log maintainer that issues inconsistent digests can be held accountable.

3. PUBLIC LOG IN DTKI

As we mentioned in the introduction, the purpose of public logs is not only to store data and make them accessible, but also to provide some cryptographically verifiable proofs on the different aspects of the log. In this section, we first describe what are the different attributes that we expect from public logs and which data structure can achieve them. Then, we describe how to use these data structure in DTKI, in particular in the case of mapping log and the certificate logs.

3.1. Data structure

In most case, when a log is used in a protocol, it is usually considered as a data collector from which we hope to retrieve some useful data when something goes wrong. As such, the implementation of the log and its underlying data structure is not considered nor described. However, the requirement for cryptographic verifiable proofs from the log forces us to specify the data structure used to model the public log and the possible actions on it. The first data structure that we consider display how the log will store the history of requests that the log maintainer received and so performed on his log. As such, this data structure, named *chronological data structure*, stores the data in the order they were added and is append-only, *i.e.* only the addition operation is allowed. We also introduce the notion of *digest* of data. Indeed, even though we do not wish for a log maintainer to send the complete content of his log to the different users, we still wish for the log maintainer to be able to send some informations about the content of its log. For this matter, we consider that each log maintainer can produce *digests* of the data it stores, in other word a short representation of the data.

Definition 3.1 (*Chronological data structure*). Let X be a set and $d \in X$. A *chronological data structure* over X is a data structure S with the following operations: $\mathsf{content}_c(S)$ that is a sequence of elements of X; $\mathsf{digest}_c(S)$ that is a value of constant size, called the *digest* of S; $\mathsf{size}_c(dg)$ that returns the number of element stored from the digest dg; $\mathsf{add}_c(S, d)$ that returns a chronological data structure. They satisfies the following properties:

— for all chronological data structures S', if $\mathsf{content}_c(S) \neq \mathsf{content}_c(S')$ then $\mathsf{digest}_c(S) \neq \mathsf{digest}_c(S')$ with overwhelming probability;
— $\mathsf{content}_c(\mathsf{add}_c(S, d)) = \mathsf{content}_c(S)$ appended with d;
— $\mathsf{size}_c(\mathsf{digest}_c(S)) = |\mathsf{content}_c(S)|$

Moreover, there exist boolean procedures $\mathsf{VerifPoP}_c$ and $\mathsf{VerifPoE}_c$, whose computation time are linear in the size of their inputs such that if we denote $dg = \mathsf{digest}_c(S)$ then

— for all $d \in X$, for all $n \in \mathbb{N}$, we have that d is the n^{th} element of $\mathsf{content}_c(S)$ if, and only if, there exists a value p of size $O(\log(|\mathsf{content}_c(S)|))$, called *proof of presence of d being the n^{th} element in dg*, such that $\mathsf{VerifPoP}_c(dg, d, n, p) = \mathsf{true}$;
— for all value dg', we have that there exists a chronological data structure S' such that $dg' = \mathsf{digest}_c(S')$ and $\mathsf{content}_c(S')$ is an initial subsequence of $\mathsf{content}_c(S)$ if, and only if, there exists a value p of size $O(\log(|\mathsf{content}_c(S)|))$, called *proof of extension of dg' into dg*, such that $\mathsf{VerifPoE}_c(dg', dg, p) = \mathsf{true}$.

The existence of digests in constant size allows us to consider them as an easily storable data by the user. Moreover, the two procedures $\mathsf{VerifPoP}_c$ and $\mathsf{VerifPoE}_c$, that must be cryptographically sound and complete, guarantee the existence of respectively proof of presence of all data in a digest and a proof of extension of a digest to another. We will denote by $\mathsf{dg}\emptyset_c$ the digest of an empty chronological data structure. Note that since the size of the proofs are logarithmic in the number of element in the data structure, the complete computation time of the procedure $\mathsf{VerifPoP}_c$ and $\mathsf{VerifPoE}_c$ are also logarithmic in the number of element in the data structure, thus making them very efficient procedure. The existence of an efficient proof of extension is crucial at least for two reasons. Firstly, it allows a log maintainer to prove to a user that he did not remove or change any data from the log since the last user's connexion. Secondly, there is no bound on how much data was added in the log between two TLS connexion of a user. Thus, it is important to define a procedure of proof of extension between two digests whose computation time does not really depend on how much data was added in the log between the two digests. For example, only using an efficient procedure verifying that a unique element was added in the log would not been enough. Even though we would have been able to define with it a procedure for verifying the extension between digests by verifying as much proofs of addition as there were data added between the two digests, it would not have given us an efficient procedure since a user would need to download and verify a number of proofs linear in the number of element added in the log.

The soundness of the procedures $\mathsf{VerifPoP}_c$ and $\mathsf{VerifPoE}_c$ rely on the knowledge that dg is a digest of some chronological data structure. This could be verified for example by a trusted third party that would download the complete data, recompute the digests and compare them. However, to avoid that a user has to trust this third party or recompute the digest himself, we introduce the notion of *randomly verifiable chronological data structure.*

Definition 3.2 (*Randomly verifiable chronological data structure*). We say that a chronological data structure is randomly verifiable if there exists a boolean procedure $\mathsf{Rand}\exists_c$, whose computation time is linear in the size of its inputs, such that:

— given a value dg and $N \in \mathbb{N}$, there exists a chronological data structure S s.t. $dg = \mathsf{digest}_c(S)$ and $N = |\mathsf{content}_c(S)|$ if, and only if, for all $n \in \{1, \dots, N\}$, there exist a value p of size $O(\log(N))$ such that $\mathsf{Rand}\exists_c(n, dg, N, p) = \mathsf{true}$;
— given two values dg, dg', given the integers $n \leq N < N'$, if there exists p_e such that $\mathsf{VerifPoE}_c(dg, dg', p_e) = \mathsf{true}$, $\mathsf{size}_c(dg) = N$ and $\mathsf{size}_c(dg') = N'$ then we have that there exists p such that $\mathsf{Rand}\exists_c(n, dg, N, p) = \mathsf{true}$ if, and only if, there exists p' such that $\mathsf{Rand}\exists_c(n, dg', N', p') = \mathsf{true}$.

Thanks to $\mathsf{Rand}\exists_{CT}$, the verification that some value is in fact a digest of a chronological data structure can be divided into several smaller verification (as much as the size of the data stored) and the computation time of each single verification is logarithmic in the size of the data. Thus, these verifications can be distributed over the users of the system hence eliminating the requirement for any monitor. Note that the second property of the definition guarantees us that the computation of $\mathsf{Rand}\exists_{CT}$ for a certain integer n is still valid, *i.e.* does not need to be recompute, for any digest supposedly representing an extension of the data structure. Even though this property is not necessary theoretically to ensure the existence of the chronological data structure associated with a digest, it makes the distributed verification much more efficient.

In a randomly verifiable chronological data structure, the procedure $\mathsf{Rand}\exists_{CT}$ gives us a tool to show that some value dg is in fact a digest of a chronological data structure. Such verification could be done of course by a monitor that would download the complete data and recompute the digest of them. But thanks to $\mathsf{Rand}\exists_{CT}$, this huge verification can be divided into several smaller verification (as much as the size of the data stored) and the computation time of each single verification is logarithmic in the size of the data. Hence, these verifications can be distributed over the users of the system thus eliminating the requirement for a monitor. Note that the second property of a randomly verifiable chronological data structure guarantees us that the computation of $\mathsf{Rand}\exists_{CT}$ for a certain integer n is still valid, *i.e.* does not need to be recomputed, for any digest supposedly representing an extension of the data structure. Even though this property is not necessary theoretically to ensure the existence of the chronological data structure associated with a digest, it makes the distributed verification much more efficient.

While the chronological data structure is a transparent, efficient and provable data structure for storing the history of requests, its append-only property makes it difficult to actually model more malleable data sets. In our case, we are interesting into storing certificates that can be revoked or can expire, and where domain name can change ownership, etc. As such, we introduce a new transparent and efficient data structure, called *ordered data structure*, that allows more operations, such as deletion and modification, and that is still able to provide cryptographically veritable proofs of any of these actions. In the following definition, given a set X partially ordered by a relation $\mathcal{R}$, we define a special element ∞ such that for all $d \in X$, $\infty \mathcal{R}\, d$ and $d\, \mathcal{R}\, \infty$. We consider such an element ∞ to express all elements in left (resp. right) relation with the minimal (resp. maximal) by $\mathcal{R}$ in X.

Definition 3.3 (*Ordered data structure*). Let X be a set partially ordered by a relation $\mathcal{R}$ and let $d, d' \in X$. An ordered data structure over X is a data structure S with the following operations: $\mathsf{content}_o(S)$ is a set of elements of X; $\mathsf{digest}_o(S)$ is a value of constant size, called the *digest* of S; $\mathsf{size}_o(dg)$ that returns the number of element stored from the digest dg; $\mathsf{add}_o(S, d)$, $\mathsf{del}_o(S, d)$ and $\mathsf{mod}_o(S, d, d')$ are respectively the addition, deletion and modification of an element, and returns an ordered data structure. They satisfies the following properties, for all $d, d' \in X$:

— for all ordered data structure S', $\mathsf{content}_o(S) \neq \mathsf{content}_o(S')$ if, and only if, $\mathsf{digest}_o(S) \neq \mathsf{digest}_o(S')$;
— $\mathcal{R}$ is a total order over D;
— $\mathsf{size}_o(\mathsf{digest}_o(S)) = |\mathsf{content}_o(S)|$;
— $\mathsf{add}_o(S, d)$ succeeds if $d \notin \mathsf{content}_o(S)$, $\mathcal{R}$ is a strict total order over $D \cup \{d\}$ and produces an ordered data structure S' such that $\mathsf{content}_o(S') = \mathsf{content}_o(S) \cup \{d\}$;
— $\mathsf{del}_o(S, d)$ succeeds if $d \in \mathsf{content}_o(S)$ and produces an ordered data structure S' such that $\mathsf{content}_o(S') = \mathsf{content}_o(S) \smallsetminus \{d\}$;
— $\mathsf{mod}_o(S, d, d')$ succeeds if $d \in \mathsf{content}_o(S)$, d and d' are $\mathcal{R}$-equivalent, and produces an ordered data structure S' such that $\mathsf{content}_o(S') = \mathsf{content}_o(S) \cup \{d'\} \smallsetminus \{d\}$.

Moreover, there exist three boolean procedures $\mathsf{VerifPoAdd}_o$, $\mathsf{VerifPoD}_o$ and $\mathsf{VerifPoM}_o$, whose computation time are linear in the size of their inputs such that if we denote $dg = \mathsf{digest}_o(S)$ and for all value dg', we have that:

— $\mathsf{add}_o(S, d)$ succeeds and $dg' = \mathsf{digest}_o(\mathsf{add}_o(S, d))$ if, and only if, there exist a value p of size $O(\log(|D|))$, called *proof of addition of d in dg into dg'*, such that $\mathsf{VerifPoAdd}_o(d, dg, dg', p) = \mathsf{true}$;
— $\mathsf{del}_o(S, d)$ succeeds and $dg' = \mathsf{digest}_o(\mathsf{del}_o(S, d))$ if, and only if, there exist a value p of size $O(\log(|D|))$, called *proof of deletion of d in dg into dg'*, such that $\mathsf{VerifPoD}_o(d, dg, dg', p) = \mathsf{true}$;
— $\mathsf{mod}_o(S, d, d')$ succeeds and $dg' = \mathsf{digest}_o(\mathsf{mod}_o(S, d, d'))$ if, and only if, there exist a value p of size $O(\log(|D|))$, called *proof of modification of d by d' in dg into dg'*, such that $\mathsf{VerifPoM}_o(d, d', dg, dg', p) = \mathsf{true}$.

At last, there exist three boolean procedures $\mathsf{VerifPoP}_o$, $\mathsf{VerifPoAbs}_o$ and $\mathsf{VerifPoAdj}_o$ whose computation time are linear in the size of their inputs such that if we denote $dg = \mathsf{digest}_o(S)$ and for all $d_1, d_2 \in X \cup \{\infty\}$, we have that:

— $d \in \mathsf{content}_o(S)$ if, and only if, there exist a value p of size $O(\log(|D|))$, called *proof of presence of d in dg*, such that $\mathsf{VerifPoP}_o(d, dg, p) = \mathsf{true}$;
— $d \notin \mathsf{content}_o(S)$ if, and only if, there exist a value p of size $O(\log(|D|))$, called *proof of absence of d in dg*, such that $\mathsf{VerifPoAbs}_o(d, dg, p) = \mathsf{true}$;
— $d_1, d_2 \in \mathsf{content}_o(S) \cup \{\infty\}$ and for all $d_0 \in X$, $d_1\, \mathcal{R}\, d_0\, \mathcal{R}\, d_2$ implies $d_0 \notin D$ if, and only if, there exist a value p of size $O(\log(|D|))$, called *proof of adjacency between d_1 and d_2 in dg*, such that $\mathsf{VerifPoAdj}_o(d_1, d_2, dg, p) = \mathsf{true}$.

We will denote by $\mathsf{dg}\emptyset_o$ the digest of an empty ordered data structure. Contrary to a chronological data structure, an ordered data structure can be freely modified by adding, deleting or even modifying data from the data structure. It is possible since the elements of the data structure are ordered by a relation $\mathcal{R}$ that only depends on the elements themselves and not the order they were added in the data structure. But more importantly, all these operations can be cryptographically proved thanks to the three associated verification procedures $\mathsf{VerifPoAdd}_o$, $\mathsf{VerifPoD}_o$ and $\mathsf{VerifPoM}_o$. As for chronological data structure, these verification procedures rely on digests of the data that are also in constant size.

Intuitively, the presence of a relation $\mathcal{R}$ over the elements of the data structure allows us to provide the three boolean procedures $\mathsf{VerifPoP}_o$, $\mathsf{VerifPoAbs}_o$ and $\mathsf{VerifPoAdj}_o$, namely the verification of presence, absence and adjacency of elements in the data structure. The verification of absence shows that one particular element in X is absent from the data structure, whereas the verification of adjacency allows us to prove that their no element in X that are in relation with two data in the data structure (d and d' in the definition) and also include in the latter, *i.e.* d and d' are adjacent in the data structure. Note that the existence of the procedure for verification of absence is guaranteed by the existence of the procedure of verification of adjacency. However, for potential efficiency reason, we preferred separated the two procedures.

For any partial order $\mathcal{R}$ over a set X such that testing whereas two elements $a, b \in X$ are in relation by $\mathcal{R}$, is efficiently computable (*i.e.* polynomial in the size of a, b), an ordered data structure can be implemented by a new data structure that we created, namely AVL hash tree (see Appendix C).

Note that an ordered data structure does not have proof of extension contrary to a chronological log. In one hand, we showed that such proof of extension was crucial for the usability and efficiency of our protocol but the proof of absence and adjacency are crucial for providing the revocation functionality to our protocol. To obtain both usability and functionality, we will combine both data structures in our log as it will be shown in the next section. Intuitively, the chronological log will be use to store the request sent to the log maintainer while ordered data structures will characterise the status of the log at the time of each requests.

All the proofs provided in Definition 3.3 allow us to cryptographically prove each small step applied on the log. Thus, when several data in the log are modified, one can prove all these modifications as long as he knows in which order they were modified. But when the order of modifications is unknown, we still wish to prove that the two digests are linked. In particular, we are interested to show that two sets of data D and D', stored in ordered data structures, are *$\mathcal{R}$-equivalent*, that is for all $d \in D$ (resp. $d \in D'$), there exists $d' \in D'$ (resp. $d \in D$) such that d and d' are $\mathcal{R}$-equivalent. As for the chronological data structure, we will rely for this on a randomise verification to prove that.

Definition 3.4 (*Modifications randomly verifiable in an ordered data structure*). We say that modifications in an ordered data structure are randomly verifiable if there exists a boolean procedure RandM_O, whose computation time is linear in the size of its inputs, such that given an ordered data structure S by $\mathcal{R}$, given a value dg', if we denote $dg = \mathsf{digest}_o(S)$ then the following two properties are equivalent:

— there exists a data structure S' such that $\mathsf{content}_o(S)$ and $\mathsf{content}_o(S')$ are $\mathcal{R}$-equivalent and $dg' = \mathsf{digest}_o(S')$;
— there exists $k \in O(\log(|\mathsf{content}_o(S)|)$ such that for all $n \in \{1, \ldots, 2^k\}$, there exists a value p of size $O(k)$, and d, d' such that d and d' are $\mathcal{R}$-equivalent and $\mathsf{RandM}_O(d, d', dg, dg', n, N, p) = \mathsf{true}$.

3.2. Mapping log

As we previously mentioned, the mapping log is used to maintain association between top-level domains and certificate log, and can provide efficient proof of current association. However, due the large amount of top-level domains existing and due to the versatile nature of domain name (*i.e.* large amount of domain are created and deleted every day), we will use regular expression as a finite representation to express the unbounded number of domain names that a log is allowed to store. Thus we will first recall the notion of regular expression, then we explain the structure of the mapping log and how it will be used in the protocol.

3.2.1. Regular expressions

Definition 3.5. A regular expression on a finite alphabet $\mathcal{A}$ is:

— the empty set $\emptyset$
— the set containing the empty string, denoted ε
— for all $a \in \mathcal{A}$, the set containing only a and denoted a
— the concatenation of two regular expressions, *i.e.* $R \cdot S = \{u \cdot v \mid u \in R \wedge v \in S\}$
— the alternation of two regular expressions, *i.e.* $R \mid S = \{u \mid u \in R \vee u \in S\}$
— the Kleene star of a regular expression, *i.e.* R^*, that is the smallest set containing the empty string and R that is closed by concatenation.

The set of regular expression on $\mathcal{A}$ is denoted $Sreg_{\mathcal{A}}$.

We can use the annotation $a?$ and $a+$ as syntactic sugar for $a \mid \varepsilon$ and $aa*$ respectively.

Example 3.1. Consider an alphabet $\mathcal{A} = \{'a','b',\ldots,'z','.','/','-''\}$. The regular expression $www.(a \mid b \mid \ldots z) + .com$ express all the domain name starting with $www.$, finishing by $.com$ and such that the middle part of the domain contains only letter from $a, \ldots, z$ but at least one. ◇

Regular expressions have been extensively studied in the literature [] and was shown to express exactly the regular languages, that is the class of languages accepted by deterministic finite automata. Thus, operations like complementary and difference are possible on regular expressions but might be expensive. Note that the notation we used in Definition 3.5 is not necessary the simplest. Traditional programming language adopted the POSIX standard to express regular expressions. For example, the POSIX standard adopts, with the already existing notions $+, ?, *, |$, the following notations

— . expresses any single character
— [ ] matches any single character in the bracket. $[a-z]$ expresses any letter of the alphabet and $[0-9]$ expresses any number between 0 and 9.
— [ˆ] matches any single character different from the ones in the bracket.
— ˆ (resp. $) indicates the beginning (resp. end) of the string

Example 3.2. Consider the alphabet $\mathcal{A}$ = {'a', 'b', …, 'z', '.' , '/', '-'}, in the POSIX notation, the regular expression $www\backslash.[a-z]*\backslash.com$ matches the same regular expression as the one in Example 3.1. ◇

The mappings between regex and domain name of certificate log maintainer will be stored in an $\mathcal{R}$-ordered log. Thus, we need to define a partial order over the regex that we will use in the mapping log. Given an alphabet $\mathcal{A}$, we will denote by $<_{\mathcal{A}}$ the strict total lexicographic order over $\mathcal{A}^*$.

Definition 3.6 (*Relation for regex*). Let $\mathcal{A}$ be an alphabet. We say that a relation $\mathcal{R}$ covers $\mathcal{A}^*$ with $R \subseteq Sreg_{\mathcal{A}}$ if:

— for all $w \in \mathcal{A}^*$, $\{w\} \subseteq R$;

— $\mathcal{R}$ is closed by transitivity;
— for all $reg_1, reg_2 \in S$, $reg_1 \, \mathcal{R} \, reg_2$ implies that for all $w_1 \in reg_1$, for all $w_2 \in reg_2$, $w_1 <_{\mathcal{A}} w_2$;
— for all $W \subseteq \mathcal{A}^*$ regular language, there exists a finite sequence $S = [reg_1, \dots, reg_n]$ of elements of R such that $\mathcal{R}$ is a total order over S and for all $w \in W$, there exists $i \in \{1, \dots, n\}$ such that $w \in reg_i$.

A relation for regex indicates how the regex will be ordered in an $\mathcal{R}$-ordered log. In the relation, R is an infinite set of regex that includes at least all singletons, *i.e.* the regex representing one word, and that is closed by transitivity. The last two properties are the heart of the definition: First, two regex in relation implies that any pairs of words in these regex are also lexicographically in relation. Second, any regular subset of $\mathcal{A}^*$ can be covered by a finite sequence of regex on which the relation is a total order. Typically, it is this sequence that will be stored in the $\mathcal{R}$-ordered log.

LEMMA 3.1. *For all alphabet $\mathcal{A}$, there exists a relation that covers $\mathcal{A}^*$ with some $R \subseteq Sreg_{\mathcal{A}}$.*

For the rest of this paper, we will denote by $\mathcal{A}$ the alphabet we use for the regex and by $\mathcal{R}_{reg}$ a relation for regex that covers $\mathcal{A}^*$.

3.2.2. Data of the mapping log. As mentioned, the mapping log will be used to store the mapping between regex and domain name of certificate log. However, we need to consider that the mapping can evolve in time, or that even the number of certificate log can change, *e.g.* a new certificate log can be added into the system. We also consider that a certificate log can be removed from the system, on in other word, can be black listed. Indeed, we consider that a certificate log cannot come and go as it pleases. Once a certificate log has been removed from the system, it cannot be reinstated anymore. At last, the mapping log will also keep informations on all *active* certificate logs to ensure synchronisation between the mapping log maintainer and all the certificate logs. To describe all possible modifications on the mapping log, we first define the set of possible request that can be sent to the mapping log maintainer.

Definition 3.7. A term is a request to a mapping log if it is of the following form:

— $\mathsf{add}(reg, id)$ or $\mathsf{del}(reg, id)$, respectively called *addition* or *deletion request*;
— $\mathsf{new}(cert, \mathsf{sign}_{sk}(n, dg, t))$ or $\mathsf{bl}(id)$ with $\mathsf{pk}(sk) = \mathsf{key}(cert)$ respectively called *new log* or *blacklist request*;
— $\mathsf{mod}(cert, \mathsf{sign}_{sk}(cert'), \mathsf{sign}_{sk'}(n, dg, t))$ with $\mathsf{pk}(sk) = \mathsf{key}(cert)$, $\mathsf{pk}(sk') = \mathsf{key}(cert')$ and $\mathsf{id}(cert) = \mathsf{id}(cert')$, called *modification request*;
— end called *end request*;

for some $reg \in Sreg_{\mathcal{A}}$, $id \in \mathcal{A}^*$, $t \in \mathbb{N}$ and $cert, cert' \in \mathcal{C}$. We denote by $\mathsf{Req_m}$ the set of requests.

We will say that the id of a request is the word $id \in \mathcal{A}^*$ in all the request but end and $\mathsf{mod}(cert, \mathsf{sign}_{sk}(cert'), t), \mathsf{new}(cert)$ where the id is $\mathsf{key}(cert)$.

The requests $\mathsf{add}(reg, id)$ and $\mathsf{del}(reg, id)$ represents respectively the addition and deletion of a mapping regex / domain name in the log. The requests $\mathsf{new}(cert, \mathsf{sign}_{sk}(n, dg, t))$ represents the insertion of a new certificate log maintainer in the system whereas $\mathsf{bl}(id)$ represents a certificate log maintainer that is being black listed. The request $\mathsf{mod}(cert, \mathsf{sign}_{sk}(cert'), \mathsf{sign}_{sk'}(n, dg, t))$ corresponds to the request done by a certificate log desiring to change or update the certificate of its public key. At last, the constant end represents the end of a sequence of requests in the mapping. Indeed, we consider that the mapping log maintainer will "emit" his modifications by block of requests, that is he will only publish the digest of his log containing the whole block of requests. To that matter, we define a relation $\leq_r$ on requests such that for all regex $reg_1, reg_2, reg_7, reg_8$, for all certificates $cert_5, cert_6$, for all words $\{id_i\}_i \in \{1 \ldots 9\}$, for all signatures $w_5, w_6, w_9, w'_9, w_{10}, w'_{10}$,

$$\begin{array}{c}\mathsf{del}(reg_1, id_1) \leq_r \mathsf{del}(reg_2, id_2) \leq_r \mathsf{bl}(id_3) \leq_r \mathsf{bl}(id_4)\\ \leq_r \mathsf{new}(cert_5, w_5) \leq_r \mathsf{new}(cert_6, w_6) \leq_r \mathsf{add}(reg_7, id_7)\\ \leq_r \mathsf{add}(reg_8, id_8) \leq_r \mathsf{mod}(cert_9, w_9, w'_9) \leq_r \mathsf{mod}(cert_{10}, w_{10}, w'_{10}) \leq_r \mathsf{end}\end{array}$$

The relation $\leq_r$ will be use later to guarantee the order of requests within the log. In the rest of this paper, we will denote by $\mathcal{C}$ the set of certificates. Moreover, given a certificate $cert \in \mathcal{C}$, we denote by $\mathsf{key}(cert)$ the public key associated to $cert$ and by $\mathsf{id}(cert)$ the name associated to $cert$. We also denote by $\mathcal{R}_\mathcal{C}$ the relation such that for all $cert, cert' \in \mathcal{C}$, $cert\,\mathcal{R}_\mathcal{C}\,cert'$ if and only if $\mathsf{id}(cert) <_\mathcal{A} \mathsf{id}(cert')$. At last, let us call $\mathcal{D}gt_c$ the finite set of possible digest values of chronological data structure, *i.e.* set of bitstrings of a particular size.

Definition 3.8. Consider the following sets and the associated relations

— $X_{st} = \{(cert, \mathsf{sign}_{sk}(dg, t)) \mid cert \in \mathcal{C}, \mathsf{pk}(sk) = \mathsf{key}(cert), dg \in \mathcal{D}gt_c \text{ and } t \in \mathbb{N}\}$ ordered by $\mathcal{R}_{st}$ that is the relation such that for all $(cert, s), (cert', s') \in S_{st}$, $(cert, s)\,\mathcal{R}_{st}(cert', s')$ if and only if $cert\,\mathcal{R}_\mathcal{C}\,cert'$;
— $X_{bl} = \mathcal{A}^*$ ordered by $<_\mathcal{A}$;
— $X_r = Sreg_\mathcal{A} \times \mathcal{A}^*$ ordered by $\mathcal{R}_r$ that is the relation such that for all $(reg, c), (reg', c') \in S_r$, $(reg, c)\,\mathcal{R}_r(reg', c')$ if and only if $reg\,\mathcal{R}_{reg}\,reg'$;
— $X_i = \{(id, \mathsf{digest}_o(S)) \mid id \in \mathcal{A}^* \text{ and } S \text{ is an ordered data structure over } Sreg_\mathcal{A}\}$ ordered by $\mathcal{R}_i$ that is the relation such that for all $(id, dg), (id', dg') \in S_i$, $(id, dg)\,\mathcal{R}_i(id', dg')$ if and only if $id <_\mathcal{A} id'$.

A *mapping log* is a chronological data structure over the set X defined as follows:

$$X = \left\{ \mathsf{h}(req, t, dg_s, dg_{bl}, dg_r, dg_i) \;\middle|\; \begin{array}{l} req \in \mathsf{Req_m}, dg_s = \mathsf{digest}_o(S_{st}), dg_{bl} = \mathsf{digest}_o(S_{bl}),\\ dg_r = \mathsf{digest}_o(S_r), dg_i = \mathsf{digest}_o(S_i),\\ S_{st}, S_{bl}, S_r, S_i \text{ are ordered data structures}\\ \text{over } X_{st}, X_{bl}, X_r, X_i \text{ respectively.}\end{array} \right\}$$

A mapping log is in fact a chronological data structure where each data stored is the request made to the mapping log maintainer with some digests of different ordered data structures representing the current status of the log at the time of the request. Thus, in $\mathsf{h}(req, t, dg_s, dg_{bl}, dg_r, dg_i) \in X$, dg_s is the digest of the data structure modelling the active certificate log maintainers: In $(cert, \mathsf{sign}_{sk}(n, dg, t)) \in X_{st}$, $cert$ is the certificate of an active certificate log maintainer, dg is the digest of its log at the time t. It is signed using the signing key associated to $cert$, *i.e.* $\mathsf{pk}(sk) = \mathsf{key}(cert)$, to ensure that it is the certificate log that provided the digest. Moreover, dg_{bl} is the digest representing the black list of certificate log maintainers. At last dg_r and dg_i both are digests of ordered data structure modelling the mapping between regular expressions and domain names, but with different indexation: X_r is ordered following the order on the regular expression whereas X_i is ordered following the order on the domain name.

3.2.3. Well formed mapping log. In our protocol, we will adopt some conventions on how a mapping log should be updated and on which properties it should satisfy. All these properties will be tested and verified by users. The reputation of the mapping log maintainer depending on the success of these tests by the users, it is essential to fully describe the properties that the mapping log should satisfy.

Addition request. A request $\mathsf{add}(reg, id)$ indicates that a new mapping regular expression / domain name of a certificate log should be added into the log. The first condition for this to be possible is that id is the domain name of a declared certificate log maintainer. The second condition is that the regular expression was not already mapped to a certificate log maintainer.

Definition 3.9. Let $\mathsf{add}(reg, id)$ be an addition request. Let $V = (t, dg_s, dg_{bl}, dg_r, dg_i)$ and $V' = (t', dg'_s, dg'_{bl}, dg'_r, dg'_i)$ be some values. We say that $\mathsf{add}(reg, id)$ *is applied on* V *into* V' if:

(1) $t = t'$, $dg_{bl} = dg'_{bl}$ and $dg_s = dg'_s$; and
(2) there exists a value p, a certificate $cert$ and a signature sig such that $\mathsf{id}(cert) = id$ and $\mathsf{VerifPoP}_o((cert, sig), dg_s, p) = \mathsf{true}$; and
(3) there exists a value p such that $\mathsf{VerifPoAdd}_o((reg, id), dg_r, dg'_r, p) = \mathsf{true}$; and
(4) one of the following two conditions holds:
  (4.a) either there exists values p and an ordered data structure S such that $\mathsf{VerifPoAdd}_o((id, dg), dg_i, dg'_i, p) = \mathsf{true}$ with $dg = \mathsf{digest}_o(S)$ and $\{reg\} = \mathsf{content}_o(S)$;
  (4.b) or else there exists values p, p', p'' and dg, dg' such that $\mathsf{VerifPoP}_o((id, dg), dg_i, p) = \mathsf{true}$, $\mathsf{VerifPoAdd}_o(reg, dg, dg', p'') = \mathsf{true}$ and $\mathsf{VerifPoM}_o((id, dg), (id, dg'), dg_i, dg'_i, p') = \mathsf{true}$.

For an addition request, the digests modelling the active and blacklisted certificate log maintainer do not change (Condition 1). On the other hand, the mapping (reg, id) is added into the ordered data structure with digest dg_r obtaining thus a data structure with digest dg'_r (Condition 3). Note that the properties of $\mathsf{VerifPoAdd}_o$ ensure us that reg was not previously mapped with and domain name. The condition 2 guarantees us that id is the domain name of a registered and active certificate log maintainer. At last, since dg_i is the digest of a data structure modelling the mappings but index by domain name, we have two different cases. In the first case, reg is the first regular expression associated with the certificate log maintainer id. In such case, id associated with a digest modelling a data structure containing only the regular expression reg, is added in the data structure with digest dg_i obtaining a data structure with digest dg'_i (Condition 4.a). In the second case, some regular expressions were already mapped to the certificate log maintainer id hence the presence of an element (id, dg) into the data structure with digest dg_i. In such a case, the entry is modified with the digest of the data structure containing reg, that is dg' (Condition 4.b).

Deletion request. A request $\mathsf{del}(reg, id)$ indicates that a mapping regular expression / domain name of a certificate log should be deleted from the log. It is the symmetric of the addition request, and so do its conditions.

Definition 3.10. Let $\mathsf{del}(reg, id)$ be a deletion request. Let $V = (t, dg_s, dg_{bl}, dg_r, dg_i)$ and $V' = (t', dg'_s, dg'_{bl}, dg'_r, dg'_i)$ be some values. We say that $\mathsf{del}(reg, id)$ *is applied on* V *into* V' if:

(1) $t = t'$, $dg_{bl} = dg'_{bl}$ and $dg_s = dg'_s$; and
(2) there exists a value p, a certificate $cert$ and a signature sig such that $\mathsf{id}(cert) = id$ and $\mathsf{VerifPoP}_o((cert, sig), dg_s, p) = \mathsf{true}$; and
(3) there exists a value p such that $\mathsf{VerifPoD}_o((reg, id), dg_r, dg'_r, p) = \mathsf{true}$; and
(4) one of the following two conditions holds:
  (4.a) there exists values p and an ordered data structure S such that $\mathsf{VerifPoD}_o((id, dg), dg_i, dg'_i, p) = \mathsf{true}$ with $dg = \mathsf{digest}_o(S)$ and $\{reg\} = \mathsf{content}_o(S)$;
  (4.b) there exists values p, p', p'' and dg, dg' such that $\mathsf{VerifPoP}_o((id, dg), dg_i, p) = \mathsf{true}$, $\mathsf{VerifPoD}_o(reg, dg, dg', p'') = \mathsf{true}$ and $\mathsf{VerifPoM}_o((id, dg), (id, dg'), dg_i, dg'_i, p') = \mathsf{true}$.

Similarly to an addition request, Condition 1 indicates that the blacklisted and active certificate log maintainers were not modified and Condition 3 indicates that the domain name id corresponds to an active and register certificate log maintainer. Symetrically to the addition request, the mapping (reg, id) is removed from the ordered data structure with digest dg_r obtaining thus a data structure with digest dg'_r (Condition 3). At last, we also have to distinguish two cases concerning the digest dg_i. In the first case, reg is the only regular expression associated with the certificate log maintainer id. In such case, the mapping (id, dg) is removed from the data structure with digest dg_i where dg is the digest of an ordered data structure containing only reg (Condition 4.a). In the second case, more than one regular expressions are associated to id, represented an element (id, dg) into the data structure with digest dg_i. In such a case, the entry is modified with the digest of the data structure free from reg, that is dg' (Condition 4.b).

New log request. A request $\mathsf{new}(cert, \mathsf{sign}_{sk}(n, dg, t_r))$ indicates that a new certificate log maintainer will be added into the system under the condition that a certificate for the domain name of the certificate log maintainer was not already registered. In the signature $\mathsf{sign}_{sk}(n, dg, t_r)$, the time t_r corresponds to the time of the mapping log maintainer at the end of the sequence of requests.

Definition 3.11. Let $\mathsf{new}(cert, \mathsf{sign}_{sk}(n, dg, t_r))$ be a new log request. Let $V = (t, dg_s, dg_{bl}, dg_r, dg_i)$ and $V' = (t', dg'_s, dg'_{bl}, dg'_r, dg'_i)$ be some values. We say that $\mathsf{new}(cert, \mathsf{sign}_{sk}(n, dg, t))$ *is applied on* V *into* V' if:

(1) $t = t' = t_r$, $dg_{bl} = dg'_{bl}$, $dg_r = dg'_r$ and $dg_i = dg'_i$; and
(2) $\mathsf{pk}(sk) = \mathsf{key}(cert)$; and
(3) there exists a value p such that $\mathsf{VerifPoAdd}_o((cert, \mathsf{sign}_{sk}(n, dg, t_r)), dg_s, dg'_s, p) = \mathsf{true}$.

Adding a new certificate log maintainer implies that no mapping was yet introduce for it. Thus, the digests dg_{bl}, dg_r and dg_i stays unchanged (Condition 1). Condition 2 ensures that the verification key of cert is the one associated with the signing key used in the signature. At last, Condition 3 guarantees that the certificate with was added in the data structure with dg_s as digest, producing a data structure with dg'_s as digest. Note that this ordered data structure is over the set X_{st} which is ordered by $\mathcal{R}_{st}$. Thus, the condition $\mathsf{VerifPoAdd}_o((cert, \mathsf{sign}_{sk}(n, dg, t)), dg_s, dg'_s, p) = \mathsf{true}$ ensures that no certificate in the data structure with digest dg_s has the same identity as $cert$.

Modification request. A request $\mathsf{mod}(cert, \mathsf{sign}_{sk}(cert'), \mathsf{sign}_{sk'}(n, dg, t_r))$ is launched when a certificate log maintainer wishes to update his old certificate $cert$ into a new one $cert'$. To ensure that the new certificates actually comes from the certificate log maintainer, the latter is required to sign the new certificate $cert'$ with his old certificate $cert$. Then the certificate log maintainer has to sign again the data representing his status, that is n, dg and t_r. Note that contrary a new log request, t_r does not correspond to the time of the mapping log maintainer at the end of the current sequence of requests but at the end of the previous sequence.

Definition 3.12. Let $\mathsf{mod}(cert, \mathsf{sign}_{sk}(cert'), \mathsf{sign}_{sk'}(n, dg, t_r))$ be a modification request. Let $V = (t, dg_s, dg_{bl}, dg_r, dg_i)$ and $V' = (t', dg'_s, dg'_{bl}, dg'_r, dg'_i)$ be some values. We say that $\mathsf{mod}(cert, \mathsf{sign}_{sk}(cert'), \mathsf{sign}_{sk'}(n, dg, t_r))$ *is applied on* V *into* V' if:

(1) $t = t'$, $dg_{bl} = dg'_{bl}$, $dg_r = dg'_r$ and $dg_i = dg'_i$; and
(2) $\mathsf{pk}(sk) = \mathsf{key}(cert)$ and $\mathsf{pk}(sk') = \mathsf{key}(cert')$; and
(3) there exists a value p such that

$$\mathsf{VerifPoM}_o((cert, \mathsf{sign}_{sk}(n, dg, t_r)), (cert', \mathsf{sign}_{sk'}(n, dg, t_r)), dg_s, dg'_s, p) = \mathsf{true}$$

Blacklist request. A request $\mathsf{bl}(id)$ is launched when a certificate log maintainer id needs to be blacklisted.

Definition 3.13. Let $\mathsf{bl}(id)$ be a blacklist request. Let $V = (t, dg_s, dg_{bl}, dg_r, dg_i)$ and $V' = (t', dg'_s, dg'_{bl}, dg'_r, dg'_i)$ be some values. We say that $\mathsf{bl}(id)$ *is applied on* V *into* V' if:

(1) $t = t'$, $dg_r = dg'_r$ and $dg_i = dg'_i$; and
(2) there exists a value p such that $\mathsf{VerifPoAbs}_o((id, \mathsf{null}), dg_i, p) = \mathsf{true}$; and
(3) there exists a value p, a certificate $cert$ and a signature sig such that $\mathsf{VerifPoD}_o((cert, sig), dg_s, dg'_s, p) = \mathsf{true}$; and
(4) there exists a value p such that $\mathsf{VerifPoAdd}_o(id, dg_{bl}, dg'_{bl}, p) = \mathsf{true}$.

Before blacklisting a certificate log maintainer, that is to add it in the data structure with digest dg_{bl} (Condition 4), it must be ensured that no mapping is still associated to the certificate log maintainer id (Condition 2) and that the certificate log maintainer was removed from the list of active certificate log maintainer (Condition 3).

Well formed mapping log. Relying on the previous definitions, we can state the notion of a well formed mapping log.

Definition 3.14. Consider a mapping log S. Let us denote $\mathsf{content}_c(S)$ by $[\mathsf{h}(req^k, t^k, dg_s^k, dg_{bl}^k, {dg_r}^k, dg_i^k)]_{k\in\{1,\dots,N\}}$. We say that a mapping log is *well formed* when $req^N = \mathsf{end}$ and for all $k \in \{1, \dots, N-1\}$,

(1) if $req^k \neq \mathsf{end}$ then $req^k \leq_r req^{k+1}$ and req^k is applied on $(t^k, dg_s^k, dg_{bl}^k, {dg_r}^k, dg_i^k)$ into $(t^{k+1}, dg_s^{k+1}, dg_{bl}^{k+1}, {dg_r}^{k+1}, dg_i^{k+1})$; and
(2) if $req^{k+1} = \mathsf{end}$ then there exists $q \in O(\log(\mathsf{size}_o(\mathsf{digest}_o(dg_s^k))))$ such that for all $j \in \{1, \dots, 2^q\}$, there exists values p, dg, dg', a certificates $cert$, a signing keys sk and $t, n, n' \in \mathbb{N}$ such that $n \leq n'$, $\mathsf{id}(cert) = \mathsf{id}(cert')$, $\mathsf{pk}(sk) = \mathsf{key}(cert)$ and:

$$\mathsf{RandM}_O((cert, \mathsf{sign}_{sk}(n, dg, t)), (cert, \mathsf{sign}_{sk}(n', dg', t^{k+1})), dg_s^k, dg_s^{k+1}, p, j, 2^q) = \mathsf{true}$$

A well formed mapping log is a mapping log where all requests have been proved to be applied on the log (Condition 1). Moreover, the sizes and digests of the active certificate logs are updated are the end of a sequence of request (Condition 2). Since these certificate logs are chronological data structure, their sizes are always increasing.

3.3. Certificate log

Thanks to the mapping log maintainer, a domain owner can easily establish which certificate log maintainer is assigned to store its certificates. Of course, upon receiving a request for registering a certificate for a domain, the certificate log maintainer (or an associated certificate authority) must assert of the identity of the requester, that is the real owner of the domain name. Since this assertion process can be costly and constraining, *e.g.* by requiring some postal confirmation, we want to limit its usage especially in the case where a domain owner would need several certificates for the same domain. Hence, we consider that each domain owner can register a unique *master signing key* that will only be used to authenticate himself to the certificate log maintainer every time he sends a request, *e.g.* when adding / revoking classical TLS certificates.

As for the mapping log, we define the set of requests that can be executed by the certificate log maintainer.

Definition 3.15. A term is a request to a certificate log if it is of the following form:

— $\mathsf{reg}(\mathsf{sign}_{sk}(cert, t, 1))$ or $\mathsf{rev}(\mathsf{sign}_{sk}(cert, t, 0))$, respectively called *registration* or *revocation request of a certificate* for some certificate $cert$ and $t \in \mathbb{N}$;
— $\mathsf{mapadd}(reg)$ or $\mathsf{mapdel}(reg)$, respectively called *addition* or *deletion of regular expression* for some regular expression reg;
— $\mathsf{upadd}(\mathsf{h}(id), h)$ or $\mathsf{updel}(\mathsf{h}(id), h)$, respectively called *addition* or *deletion update*, for some domain name id and some hash value h.

We denote by $\mathsf{Req_c}$ the set of requests.

The request $\mathsf{reg}(\mathsf{sign}_{sk}(cert, t, 1))$ indicates the registration of a certificate. The signing key sk corresponds to the current master key. The integer t represents the time this request was made and is used for accountability. The constant $\mathsf{1}$ (and $\mathsf{0}$ for the revocation request) is used to prevent a signature to be used in both revocation and registration request. Note that when a (new) domain owner registers for the first time its master key, it should be self signed. Moreover, it is precisely at that time that the external verification done by the log maintainer (or external certificate authority) should occur. When a certificate for a master key is added and signed with the current master key, this new certificate replace the current certificate for master key. The request $\mathsf{rev}(\mathsf{sign}_{sk}(cert, t, 0))$ allows a domain owner to revoke its certificate, including the certificate of the master key, *e.g.* at the end of its ownership of the domain. Lastly, the requests $\mathsf{upadd}(\mathsf{h}(id), h)$ or $\mathsf{updel}(\mathsf{h}(id), h)$ are in fact self-applied by the certificate log maintainer when the mapping log maintainer changed the mappings associated to this certificate log maintainer.

Definition 3.16. Consider X_{id} the set of elements of the form $(\mathsf{h}(id), \mathsf{h}(cert, dg_a, dg_r))$ where $id \in \mathcal{A}^*$, $cert \in \mathcal{C}$, $dg_a = \mathsf{digest}_o(S_a)$, $dg_r = \mathsf{digest}_o(S_r)$, and S_a, S_r are ordered data structure over $\mathcal{C}$. Consider X_{reg} the set of elements of the form (reg, dg_{id}) where $reg \in Sreg_A$ and $dg_{id} = \mathsf{digest}_o(S_{id})$ for some ordered data structure over X_{id}.

A *certificate log* is a chronological data structure over the following set:

$$\left\{ \mathsf{h}(req, N, dg) \;\middle|\; \begin{array}{l} req \in \mathsf{Req_c}, N \in \mathbb{N}, dg \in \mathsf{digest}_o(S), \\ S \text{ ordered data structure over } X_{reg} \end{array} \right\}$$

Each request of the chronological data structure in the certificate log is accompanied by the size of the mapping log at the time of the request, *i.e* N, and a digest of an ordered data structure matching regexes with the certificates stored in the log and instance of this regex, *i.e.* (reg, dg_{id}) where dg_{id} is the digest of the ordered data structure storing the certificates. In the element $(\mathsf{h}(id), \mathsf{h}(cert, dg_a, dg_r))$, id corresponds to the domain name, $cert$ is the current certificate of the master key, dg_a (resp. dg_r) is the digest of an ordered data structure storing the active (resp. revoked) certificates. This organisation of the data allows one to easily determine (and obtain the corresponding cryptographic proofs) which domain names are stored under a particular regular expression in the certificate log, or which active or revoked certificates are associated to a domain name. Note that by keeping the revoked certificate, a domain owner can easily determines if a master signing key was compromised and used to create or revoked fake certificates.

3.3.1. Well formed certificate log. As it was the case in the for the mapping log, the digests denoted in the previous definition by dg_a, dg_r, dg, dg_{id} represents the status of the certificate log current to the request itself. Hence, the digests of an entry in the chronological data structure are the results of the application of the request req on the digests of the previous entry stored in the log.

Registration request. The application of a registration request $\mathsf{reg}(\mathsf{sign}_{sk}(cert, t, 1))$ on the log depends on the type of the certificate $cert$, that is TLS or master certificate. In both case, it must be signed by the signing key of the master certificate logged for the domain name. If $cert$ is a TLS certificate then $cert$ is just added to the log else if $cert$ is a master key, the registration request will replace the logged master key by $cert$. Note that a master certificate and TLS certificate cannot be added again once they have been revoked.

Definition 3.17. Let $\mathsf{reg}(\mathsf{sign}_{sk}(cert, t, 1))$ be a registration request where $cert$ is a master certificate. Let dg and dg' be two values. We say that $\mathsf{reg}(\mathsf{sign}_{sk}(cert, t, 1))$ *is applied on dg into dg'* if there exist values $p_1, p_2, dg_{id}, dg'_{id}, id$ and a regular expression reg such that $\mathsf{VerifPoP}_o((reg, dg_{id}), dg, p_1) = \mathsf{true}$, $id = \mathsf{id}(cert) \in reg$, $\mathsf{VerifPoM}_o((reg, dg_{id}), (reg, dg'_{id}), dg, dg', p_2) = \mathsf{true}$ and:

(1) either there exists values p_3 and dg'_{id} such that $\mathsf{VerifPoAdd}_o((\mathsf{h}(id), \mathsf{h}(cert, \mathsf{null}, \mathsf{null})), dg_{id}, dg'_{id}, p_3) = \mathsf{true}$ and $\mathsf{pk}(sk) = \mathsf{key}(cert)$;

(2) or else there exists some values p_3, p_4, p_5, dg_a, dg'_a, dg_r and a master certificate $cert_m$ such that
  (2.a) $\mathsf{VerifPoAbs}_o(cert, dg_r, p_3) = \mathsf{true}$; and
  (2.b) if $cert_m \neq \mathsf{null}$ then $\mathsf{VerifPoAdd}_o(cert_m, dg_r, dg'_r, p_4) = \mathsf{true}$ else $dg'_r = dg_r$; and
  (2.c) $\mathsf{VerifPoM}_o(d, d', dg_{id}, dg'_{id}, p_5) = \mathsf{true}$; and
  (2.d) either $cert_m$ is expired at time t and $\mathsf{pk}(sk) = \mathsf{key}(cert)$; or else $cert_m = \mathsf{null}$ and $\mathsf{pk}(sk) = \mathsf{key}(cert)$; or else $\mathsf{pk}(sk) = \mathsf{key}(cert_m)$;
  where $d = (\mathsf{h}(id), \mathsf{h}(cert_m, dg_a, dg_r))$, $d' = (\mathsf{h}(id), \mathsf{h}(cert, dg'_a, dg'_r))$ and $dg'_a = dg_a$ if $\mathsf{pk}(sk) = \mathsf{key}(cert_m)$ else $dg'_a = \mathsf{null}$.

A master certificate $cert$ can be registered under several conditions. In particular, there must exists a regular expression already registered to which $\mathsf{id}(cert)$ is an instance, *i.e.* $\mathsf{VerifPoP}_o((reg, dg_{id}), dg, p_1) = \mathsf{true}$. The digest dg_{id} will then be adapted into the new digest $dg_{id'}$ of a data structure containing $cert$, *i.e.* $\mathsf{VerifPoM}_o((reg, dg_{id}), (reg, dg'_{id}), dg, dg', p_2) = \mathsf{true}$. To describe the link between dg_{id} and dg'_{id}, we distinguish two cases. In the first case, $cert$ is the first master certificate for the domain $id = \mathsf{id}(cert)$ to be added in the log, also implying that no TLS certificate has been logged for id too (Condition 1). In the second case, a master key (and potentially some TLS certificates) are already logged. In such a case, registering a master certificate $cert$ is only possible if $cert$ has not already been revoked (Condition 2.a) and if the current master certificate $cert_m$ is either expired, or was revoked or was used to sign the request (Condition 2.d). Moreover, when $cert_m$ is not already revoked, it must be added to the data structure storing the revoked certificates (Condition 2.b). Condition 2.c establishes how dg'_{id} is generated by updating the entry for the domain name id.

Definition 3.18. Let $\mathsf{reg}(\mathsf{sign}_{sk}(cert, t, 1))$ be a registration request where $cert$ is a TLS certificate. Let dg and dg' be two values. We say that $\mathsf{reg}(\mathsf{sign}_{sk}(cert, t, 1))$ *is applied on dg into dg'* if there exist values $p_1, p_2, p_3, p_4, dg_{id}, dg'_{id}, dg_r, dg_a, dg'_a$, a domain name id, a master certificate $cert_m$ and a regular expression reg such that $id = \mathsf{id}(cert) \in reg$ and:

(1) $\mathsf{VerifPoM}_o((reg, dg_{id}), (reg, dg'_{id}), dg, dg', p_1) = \mathsf{true}$; and
(2) $cert_m$ is not expired at time t and $\mathsf{pk}(sk) = \mathsf{key}(cert_m)$; and
(3) $\mathsf{VerifPoAbs}_o(cert, dg_r, p_2) = \mathsf{true}$; and
(4) $\mathsf{VerifPoAdd}_o(cert, dg_a, dg'_a, p_3) = \mathsf{true}$; and
(5) $\mathsf{VerifPoM}_o((\mathsf{h}(id), \mathsf{h}(cert_m, dg_a, dg_r)), (\mathsf{h}(id), \mathsf{h}(cert_m, dg'_a, dg_r)), dg_{id}, dg'_{id}, p_4) = \mathsf{true}$.

As for the case of a request for registering a master certificate, a TLS certificate can only be registered if there exists a regular expression reg already registered for which $\mathsf{id}(cert)$ is an instance (Condition 1). Moreover, a master certificate $cert_m$ must also be logged for the domain name $id = \mathsf{id}(cert)$ with some digest dg_a and dg_r respectively digest of data structure storing the active and revoked certificates for this domaine. In that case, the registration can only be proceed if $cert_m$ is not expired and was used to sign the request

(Condition 2) and if *cert* was not already revoked (Condition 3). At last, Conditions 4 and 5 indicates how the digests are updated to include the new TLS certificate *cert*.

Revocation request. Similarly to the registration request, the application of a revocation request $\mathsf{rev}(\mathsf{sign}_{sk}(cert, t, 0))$ depends on the type of the certificate *cert*, that is master or TLS certificate.

Definition 3.19. Let $\mathsf{rev}(\mathsf{sign}_{sk}(cert, t, 0))$ be a revocation request where *cert* is a TLS certificate. Let dg and dg' be two values. We say that $\mathsf{rev}(\mathsf{sign}_{sk}(cert, t, 0))$ *is applied on dg into dg'* if there exist values $p_1, p_2, p_3, p_4, dg_{id}, dg'_{id}, dg_r, dg'_r, dg_a, dg'_a$, a domain name id, a master certificate $cert_m$ and a regular expression reg such that $id = \mathsf{id}(cert) \in reg$, $cert_m$ is not expired at time t, $\mathsf{pk}(sk) = \mathsf{key}(cert_m)$, $\mathsf{VerifPoM}_o((reg, dg_{id}), (reg, dg'_{id}), dg, dg', p_1) = \mathsf{true}$, $\mathsf{VerifPoAdd}_o(cert, dg_r, dg'_r, p_2) = \mathsf{true}$ and:

(1) if $cert = cert_m$ then

$$\mathsf{VerifPoM}_o((\mathsf{h}(id), \mathsf{h}(cert_m, \mathsf{null}, dg_r)), (\mathsf{h}(id), \mathsf{h}(\mathsf{null}, \mathsf{null}, dg'_r)), dg_{id}, dg'_{id}, p_3) = \mathsf{true}$$

(2) else
 (2.a) $\mathsf{VerifPoD}_o(cert, dg_a, dg'_a, p_3) = \mathsf{true}$; and
 (2.b) $\mathsf{VerifPoM}_o((\mathsf{h}(id), \mathsf{h}(cert_m, dg_a, dg_r)), (\mathsf{h}(id), \mathsf{h}(cert_m, dg'_a, dg'_r)), dg_{id}, dg'_{id}, p_4) = \mathsf{true}$.

In all cases, applying the revocation request $\mathsf{rev}(\mathsf{sign}_{sk}(cert, t, 0))$ requires that sk to be the key of the active master certificate with the same domain name as *cert*, *i.e.* $cert_m$ where $(\mathsf{h}(id), \mathsf{h}(cert_m, dg_a, dg_r))$ is the entry dedicated to the certificates of id. Moreover, since *cert* will become revoked, it has to be stored in the data structure storing the revoked certificates, that is $\mathsf{VerifPoAdd}_o(cert, dg_r, dg'_r, p_2) = \mathsf{true}$. In case, *cert* is in fact the master certificate $cert_m$, we require that all the TLS certificates were previously revoked, *i.e.* the digest dg_a is in fact the digest of an empty data structure, thats is null (Condition 1). Otherwise, *cert* is a TLS certificate and it will be removed from the data structure storing the active certificate (Condition 2.a).

Addition and deletion of regular expression. A request of addition (resp. deletion) of a regular expression $\mathsf{mapadd}(reg)$ (resp $\mathsf{mapdel}(reg)$) is applied when the mapping log maintainer requested some changes for the mapping between domain names and certificate log maintainers. This request must be applied before addition (resp. after deletion) of any certificate whose domain name is an instance of reg.

Definition 3.20. Let $\mathsf{mapadd}(reg)$ be a request for adding a regular expression. Let dg and dg' be two values. We say that $\mathsf{mapadd}(reg)$ *is applied on dg into dg'* if there exists p such that $\mathsf{VerifPoAdd}_o((reg, \mathsf{null}), dg, dg', p) = \mathsf{true}$.

Let $\mathsf{mapdel}(reg)$ be a request for deleting a regular expression. We say that $\mathsf{mapdel}(reg)$ *is applied on dg into dg'* if there exists p such that $\mathsf{VerifPoD}_o((reg, \mathsf{null}), dg, dg', p) = \mathsf{true}$.

Addition and deletion update. A request of addition (resp. deletion) update $\mathsf{upadd}(\mathsf{h}(id), h)$ (resp $\mathsf{updel}(\mathsf{h}(id), h)$) is also a request that is applied when the mapping log maintainer request some changes for the mapping between domain names and certificate log maintainers. Typically, once the certificate log maintainers adds the regular expression he has to take care of, he adds the different certificates of the domain names instance of this regular expression by using the request of addition update.

Definition 3.21. Let $\mathsf{upadd}(\mathsf{h}(id), h)$ be a request of addition update. Let dg and dg' be two values. We say that $\mathsf{upadd}(\mathsf{h}(id), h)$ *is applied on dg into dg'* if there exists values $p_1, p_2, dg_{id}, dg'_{id}$ and a regular expression reg such that $id \in reg$, $\mathsf{VerifPoAdd}_o(\mathsf{h}(id), h), dg_{id}, dg'_{id}, p_1) = \mathsf{true}$ and $\mathsf{VerifPoM}_o((reg, dg_{id}), (reg, dg'_{id}), dg, dg', p_2) = \mathsf{true}$.

Let $\mathsf{updel}(\mathsf{h}(id), h)$ be a request of deletion update. Let dg and dg' be two values. We say that $\mathsf{updel}(\mathsf{h}(id), h)$ *is applied on dg into dg'* if there exists values $p_1, p_2, dg_{id}, dg'_{id}$ and a regular expression reg such that $id \in reg$, $\mathsf{VerifPoD}_o(\mathsf{h}(id), h), dg_{id}, dg'_{id}, p_1) = \mathsf{true}$ and $\mathsf{VerifPoM}_o((reg, dg_{id}), (reg, dg'_{id}), dg, dg', p_2) = \mathsf{true}$.

Well formed certificate log. Relying on the previous definitions, we can state the notion of a well formed certificate log.

Definition 3.22. Consider a certificate log S. Let us denote $\mathsf{content}_c(S)$ by $[\mathsf{h}(req_k, n_k, dg_k)]_{k \in \{1,\dots,N\}})$. We say that a certificate log is *well formed* when for all $k \in \{1, \dots, N-1\}$, $n_k \leq n_{k+1}$ and reg_k is apply on dg_k into dg_{k+1}.

As described in the definition, the two conditions for a certificate log to be well formed is that all its requests has been proven applied and that the integer representing the size of the mapping log is always increasing, that is $n_1 \leq \ldots \leq n_N$.

3.4. Synchronisation between certificate and mapping logs

We've described the notions of well formed certificate log and mapping log. Even though both notions describe how the requests of a log are properly applied, they do not describe the link between the different certificate logs and the mapping log. Indeed, when the mapping log maintainer applies a sequence of requests in the mapping log, the addition update and deletion update applied by a certificate log depend in fact on the certificates that were registered in other certificate logs. To express this link between the different logs, we define the notion of *synchronisation* between certificate and mapping logs. For this purpose, we first introduce the notion of a domain name being blacklisted in the digest of a mapping log.

Definition 3.23. Let dg_m be some digest. Let id_c be some domain name. We say that *id_c is blacklisted in dg_m* if there exists $req_m \in \mathsf{Req_m}$, some values $dg_s, dg_{bl}, dg_r, dg_i, p, p'$ and an integer t such that $\mathsf{VerifPoP}_c(\mathsf{h}(req_m, t, dg_s, dg_{bl}, dg_r, dg_i), dg_m, \mathsf{size}_c(dg_m), p) = \mathsf{true}$ and $\mathsf{VerifPoP}_o(id_c, dg_{bl}, p') = \mathsf{true}$.

In Definition 3.14, we saw that in a well formed mapping log, there is no possibility for a certificate log maintainer to be removed from the data structure storing the blacklisted certificate log maintainers. Hence, a domain name id_c is blacklisted in dg_m if id_c occurs as blacklisted (*i.e.* $\mathsf{VerifPoP}_o(id_c, dg_{bl}, p') = \mathsf{true}$) in the latest entry of the mapping log, *i.e.* $\mathsf{VerifPoP}_c(\mathsf{h}(req_m, t, dg_s, dg_{bl}, dg_r, dg_i), dg_m, \mathsf{size}_c(dg_m), p) = \mathsf{true}$.

Secondly, when the mapping log maintainer applies an addition request (resp. deletion request) for some certificate log maintainer id, an equivalent request should be found in the certificate log of id, that is an mapping addition request (resp. mapping deletion request), independently of the TLS and master certificates stored by the certificate log maintainers. Of course, when the certificate log maintainer id is blacklisted in the mapping log, we do not expect it to respect the mapping log since it could have been found malicious or it could just be offline. This property is given in the following definition.

Definition 3.24. Let dg_m and $dg_{c_1}, \dots, dg_{c_n}$ be some values. Let $id_{c_1}, \dots, id_{c_n}$ be some domain names. We say that dg_m is respected by $(id_{c_1}, dg_{c_1}), \dots, (id_{c_n}, dg_{c_n})$ if for all $i \in \{1, \dots, \mathsf{size}_c(dg_m)\}$, there exist $req \in \mathsf{Req_m}$, an integer t, and some values $dg_s, dg_{bl}, dg_r, dg_i$, and p_1, p_2 such that:

(1) $\mathsf{VerifPoP}_c(\mathsf{h}(req, t, dg_s, dg_{bl}, dg_r, dg_i), dg_m, i, p_1) = \mathsf{true}$; and

(2) if $req = \mathsf{add}(reg, id)$ then either id is blacklisted in dg_m or else there exist $j \in \{1\ldots, n\}$ and $k \in \mathbb{N}$ such that $id = id_{c_j}$ and $\mathsf{VerifPoP}_c(\mathsf{h}(\mathsf{mapadd}(reg), i, dg), dg_{c_j}, k, p_2) = \mathsf{true}$; and
(3) if $req = \mathsf{del}(reg, id)$ then either id is blacklisted in dg_m or else there exist $j \in \{1\ldots, n\}$ and $k \in \mathbb{N}$ such that $id = id_{c_j}$ and $\mathsf{VerifPoP}_c(\mathsf{h}(\mathsf{mapdel}(reg), i, dg), dg_{c_j}, k, p_2) = \mathsf{true}$.

We can now state the notion of synchronisation between the mapping log and the different certificate logs. Intuitively, it verifies that each request of a certificate log was authorised by the mapping log, whether it is a client request such as registration or revocation of a TLS/master certificate, or a mapping request that the certificate log maintainer are complied to apply when the mapping log maintainer modifies the mapping of domain names. In the case of an addition update request, it also has to verify that the certificates added were stored by the certificate log maintainer in charge of the certificates before the changes in the mapping of domain names.

Definition 3.25. Let dg_m and $dg_{c_1}, \ldots, dg_{c_n}$ be some values. Let $id_{c_1}, \ldots, id_{c_n}$ be some domain names. We say that $(id_{c_1}, dg_{c_1}), \ldots, (id_{c_n}, dg_{c_n})$ *synchronise with* dg_m if for all $i \in \{1, \ldots, n\}$, for all $j \in \{1, \ldots, \mathsf{size}_c(dg_{c_i})\}$,

(1) dg_m is satisfied by $(id_{c_1}, dg_{c_1}), \ldots, (id_{c_n}, dg_{c_n})$; and
(2) $\mathsf{VerifPoP}_c(\mathsf{h}(req_c, N, dg), dg_{c_i}, j, p_1) = \mathsf{true}$; and
(3) $\mathsf{VerifPoP}_c(\mathsf{h}(req_m, t_m, dg_s, dg_{bl}, dg_r, dg_i), dg_m, N, p_2) = \mathsf{true}$; and

Conditions when req_c is an registration or revocation request, that is:

(4) if $req_c = \mathsf{reg}(\mathsf{sign}_{sk}(cert, t, 1))$ or $req_c = \mathsf{rev}(\mathsf{sign}_{sk}(cert, t, 0))$ then $req_m = \mathsf{end}$ and
 (4.a) there exists a certificate $cert_c$, a signing key sk_c, some values p, dg' and two integers m, t' such that $\mathsf{VerifPoP}_o((cert_c, \mathsf{sign}_{sk_c}(m, dg', t')), dg_s, p) = \mathsf{true}$, $m < j$ and $\mathsf{id}(cert_c) = id_{c_i}$; and
 (4.b) there exists a regular expression reg and value p' such that $\mathsf{VerifPoP}_o((reg, id_{c_i}), dg_r, p') = \mathsf{true}$ and $\mathsf{id}(cert) \in reg$.

Conditions when req_c is an addition or deletion mapping request, that is:

(5) if $req_c = \mathsf{mapadd}(reg)$ (resp. $\mathsf{mapdel}(reg)$) then $req_m = \mathsf{add}(reg, id_{c_i})$ (resp. $\mathsf{del}(reg, id_{c_i})$)

Conditions when req_c is an addition or deletion update request, that is:

(6) if $req_c = \mathsf{upadd}(\mathsf{h}(id), h)$ (resp. $\mathsf{updel}(\mathsf{h}(id), h)$) then there exists $req'_m \in \mathsf{Req_m}$, two regular expressions reg, reg', some values $dg'_s, dg'_{bl}, dg'_r, dg'_i$, a domain name id_c and two integer N', t'_m such that:
 (6.a) $req_m = \mathsf{add}(reg, id_{c_i})$ (resp. $\mathsf{del}(reg, id_{c_i})$ and $id \in reg$; and
 (6.b) $req'_m = \mathsf{del}(reg', id_c)$ (resp $\mathsf{add}(reg', id_c)$), $id \in reg'$, $t_m = t'_m$ and $\mathsf{VerifPoP}_c(\mathsf{h}(req'_m, t'_m, dg'_s, dg'_{bl}, dg'_r, dg'_i), dg_m, N', p_3) = \mathsf{true}$; and
 (6.c) either id_c is blacklisted in dg_m or else there exists $k \in \{1, \ldots, n\}$, an integer N'', two values dg', p_4 and $req' \in \mathsf{Req_c}$ such that $id_c = id_{c_k}$, $req' = \mathsf{updel}(\mathsf{h}(id), h)$ (resp. $\mathsf{upadd}(\mathsf{h}(id), h)$) and $\mathsf{VerifPoP}_c(\mathsf{h}(req', N', dg'), dg_{c_k}, N'', p_4) = \mathsf{true}$.

In Definition 3.25, the $(id_{c_1}, dg_{c_1}), \ldots, (id_{c_n}, dg_{c_n})$ represent typically all the active certificate log maintainers in the mapping log with digest dg_m. As we previously mentioned, we consider that the mapping log and the certificates log maintainers are synchronised when each request req_c of each certificate logs (Condition 2) are authorised by some request req_m in the mapping log (Condition 3). Depending of the request req_c, the conditions on req_m and the mapping log are different. In particular, if req_c is a registration or revocation request, that is a client request, then we check that req_m corresponds to an end request end in which we can verify the status of the certificate log maintainer (Condition 4.a) and also that the latter is authorised to handle this client request (Condition 4.b). If req_c is an addition or deletion mapping request, we verify that this addition or deletion is actually requested by the mapping log maintainer (Condition 5). The case where req_c is an addition update request is illustrated in Figure 1. Typically, to an addition update request $\mathsf{upadd}(\mathsf{h}(id), h)$ in the certificate log with N as size for the mapping log should correspond an addition request in the mapping log capturing id (Condition 6.a). But $(\mathsf{h}(id), h)$ corresponds to some already registered certificates for id, meaning that they were stored in some certificate log maintainer that was authorised by the mapping log (Condition 6.c). Since certificates for a domain name can only be stored by one certificate log maintainer at a given time, it also implies that the mapping log must have issue a deletion request for this mapping (Condition 6.b).

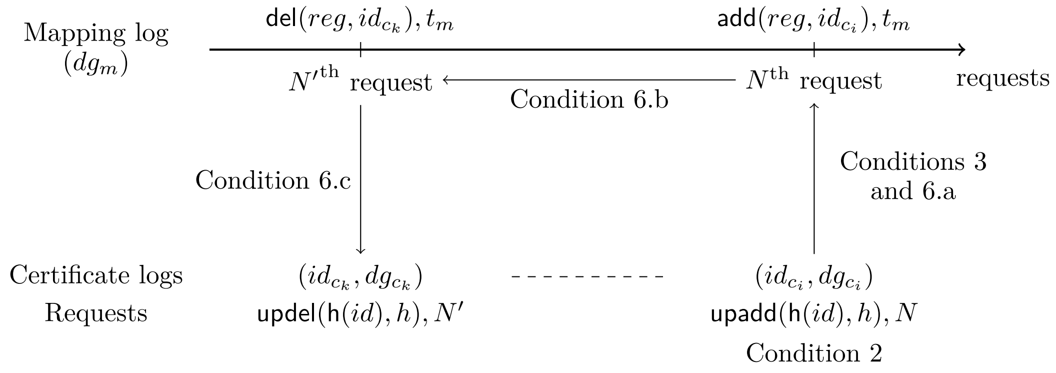


Fig. 1. Synchronisation between mapping and certificate logs: the case of addition update request.

4. DISTRIBUTED TRANSPARENT KEY INFRASTRUCTURE

Distributed transparent key infrastructure (DTKI) contains three main phases, namely certificate publication, certificate verification, and log verification. In the certificate publication phase, domain owners can upload new certificates and revoke existing certificates in the certificate log they are assigned to; in the certificate verification phase, one can verify the validity of a certificate; and in the log verification phase, one can verify whether a log behaves correctly.

We present DTKI using the scenario that a TLS user Alice wants to securely communicate with a domain owner Bob who maintains the domain *example.com*.

4.1. Certificate publication

To publish or revoke certificates in the certificate log, the domain owner Bob needs to know which certificate log is currently authorised to record certificates for his domain. This can be done by communicating with a mirror of the mapping log. We detail the protocol for requesting the mapping for Bob's domain.

4.1.1. Request mappings. Bob starts by sending a request with his domain name to a mirror of the mapping log. Upon receiving the request, the mirror locates the certificate *cert* of the authorised certificate log maintainer and generates the proofs that will be verified by Bob. To do so, the mirror obtains the data of the latest element of its copy of the mapping log, denoted $h = \mathsf{h}(req, t, dg^s, dg^{bl}, dg^r, dg^i)$, and generates the proof of its presence in the digest (denoted dg_{mlog}) of its log of size N. Then, it generates the proof of presence ofc the element $(cert, \mathsf{sign}_{sk}(n, dg, t))$ in the digest dg^s for some $\mathsf{sign}_{sk}(n, dg, t)$, proving that the certificate log maintainer whose *cert* belongs to is still active. Moreover, it generates the proof of presence of some element (rgx, id) in the digest dg^r where id is the subject of $cert$ and $example.com$ is an instance of the regular expression rgx, proving that id is authorised to stores the certificates of $example.com$. The mirror then sends to Bob the different digests $dg^s, dg^{bl}, dg^r, dg^i$, the time t, the request req, the signature $\mathsf{sign}_{sk}(n, dg, t)$, the regular expression rgx, the three generated proofs of presence, and the latest *signed Mlog timestamp* containing the time t_{Mlog}, and digest dg_{mlog} and size N of the mapping log.

Bob first verifies the received *signed Mlog timestamp* with the public key of the mapping log maintainer embedded in the browser, and verifies whether t_{Mlog} is valid. Then Bob checks that $example.com$ is an instance of rgx, verifies that $req = \mathsf{end}$ and verifies the three different proofs of presence. If all checks hold, then Bob sends the *signed Mlog timestamp* containing $(t'_{Mlog}, dg'_{mlog}, N')$ that he stored during a previous connection, and expects to receive a proof of extension of (dg'_{mlog}, N') into (dg_{mlog}, N). If the received proof of extension is valid, then Bob stores the current *signed Mlog timestamp*, and believes that the certificate log with identity id, certificate $cert$, and size that should be no smaller than n, is currently authorised for managing certificates for his domain.

4.1.2. Certificate publication. The first time Bob wants to publish a certificate for his domain, he needs to generate a pair of master signing key, denoted sk_m, and verification key. The latter is sent to a certificate authority, which verifies Bob's identity and issues a master certificate $cert_m$ for Bob. After Bob receives his master certificate, he checks the correctness of the information in the certificate. The TLS certificate can be obtained in the same way.

Figure 2 presents the process to publish the master certificate $cert_m$. Bob signs the certificate together with the current time t by using the master signing key sk_m, and sends it together with the request to the authorised certificate log maintainer whose signing key is denoted sk_{clog}. The certificate log maintainer checks whether there exists a valid master certificate for $example.com$; if there is one, then the log maintainer aborts the conversation. Otherwise, the log maintainer verifies the validity of time t and the signature.

If they are all valid, the log maintainer updates the log, generates the proof of presence of $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id}, (rgx, dg^{id}) in dg^{rgx}, and $\mathsf{h}(\mathsf{reg}(sign_{sk_m}(cert_m, t)), N_{mlog}, dg^{rgx})$ is the last element in the data structure represented by dg_{clog}, where id is the subject of $cert_m$ and an instance of rgx; $\mathsf{reg}(sign_{sk_m}(cert_m, t))$ is the register request to adding $cert_m$ into the certificate log with digest dg_{clog} at time t. The log maintainer then issues a signature on (dg_{clog}, N, h), where N is the size of the certificate log, and $h = \mathsf{h}((rgx, dg^{id}), dg^{rgx}, P)$, where P is the sequence of the generated proofs, and sends the signature σ_2 together with $(dg_{clog}, N, rgx, dg^{id}, dg^{rgx}, dg^a, dg^{rv}, P)$ to Bob. If the signature and the proof are valid, and N is no smaller than the size n contained in the *signed Mlog timestamp* that Bob received from the mirror, then Bob stores the signed (dg_{clog}, N, h), sends the previous stored (dg'_{clog}, N') to the certificate log maintainer, and expects to receive a proof of extension of (dg'_{clog}, N') into (dg_{clog}, N). If the received proof of extension is valid, then Bob believes that he has successfully published the new certificate.

Note that it is important to send (dg'_{clog}, N') after receiving (dg_{clog}, N), because otherwise the log maintainer could learn the digest that Bob has, then give a pair (dg''_{clog}, N'') of digest and size of the log such that $N' < N'' < N$. This may open a window to attackers who wants to convince Bob to use a certificate which was valid in dg''_{clog} but revoked in dg_{clog}.

The process of adding a TLS certificate is similar to the process of adding a master certificate, but the log maintainer needs to verify that the TLS certificate is signed by the valid master signing key corresponding to the master certificate in the log. The process of adding a certificate revocation request is also similar to the process of adding a new certificate. However, for a revocation request with $\mathsf{sign}_{sk_m}(cert, t)$, the log maintainer needs to additionally check that $\mathsf{sign}_{sk_m}(cert, t')$ is already in the log and $t > t'$.

4.2. Certificate verification

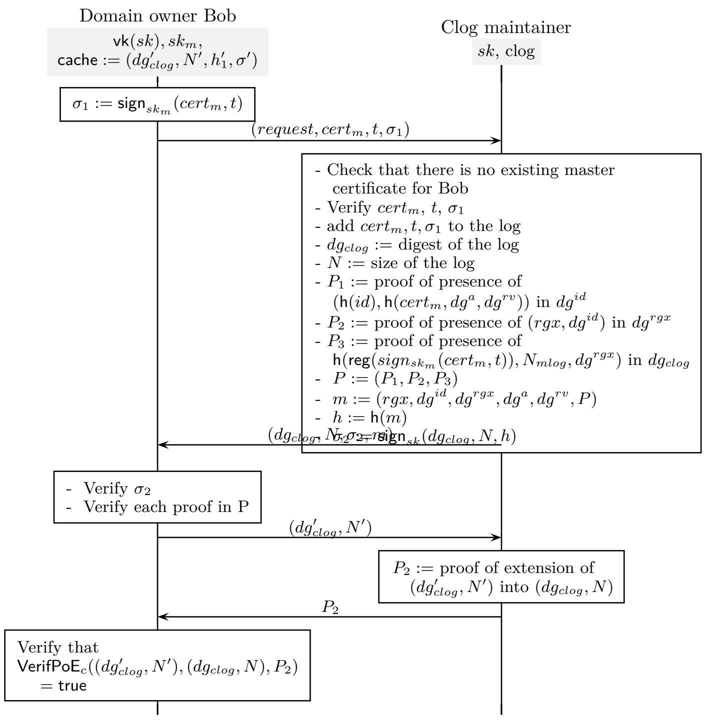

Domain owner Bob
$\mathsf{vk}(sk), sk_m,$
$\mathsf{cache} := (dg'_{clog}, N', h'_1, \sigma')$
Clog maintainer
sk, clog
$\sigma_1 := \mathsf{sign}_{sk_m}(cert_m, t)$
$(request, cert_m, t, \sigma_1)$
- Check that there is no existing master certificate for Bob
- Verify $cert_m$, t, σ_1
- add $cert_m, t, \sigma_1$ to the log
- dg_{clog} := digest of the log
- N := size of the log
- P_1 := proof of presence of $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id}
- P_2 := proof of presence of (rgx, dg^{id}) in dg^{rgx}
- P_3 := proof of presence of $\mathsf{h}(\mathsf{reg}(sign_{sk_m}(cert_m, t)), N_{mlog}, dg^{rgx})$ in dg_{clog}
- $P := (P_1, P_2, P_3)$
- $m := (rgx, dg^{id}, dg^{rgx}, dg^a, dg^{rv}, P)$
- $h := \mathsf{h}(m)$
- $\sigma_2 := \mathsf{sign}_{sk}(dg_{clog}, N, h)$
$(dg_{clog}, N, \sigma_2, m)$
- Verify σ_2
- Verify each proof in P
(dg'_{clog}, N')
P_2 := proof of extension of (dg'_{clog}, N') into (dg_{clog}, N)
P_2
Verify that
$\mathsf{VerifPoE}_c((dg'_{clog}, N'), (dg_{clog}, N), P_2)$
$= \mathsf{true}$

Fig. 2. The protocol presenting how domain owner Bob communicates with certificate log (clog) maintainer to publish a master certificate $cert_m$.

When Alice wants to securely communicate with *example.com*, she sends the connection request to Bob, and expects to receive a master certificate $cert_m$ and a signed TLS certificate $\mathsf{sign}_{sk_m}(cert, t)$ from him. To verify the received certificates, Alice checks whether the certificates are expired. If both of them are still in the validity time period, Alice requests (as described in 4.1.1) the corresponding mapping from a mirror to find out the authorised certificate log for *example.com*, and communicates with the authorised certificate log maintainer to verify the received certificate.

The Fig. 3 presents the process of verifying a certificate. After Alice learns the identity of the authorised certificate log, she sends the verification request with her local time t_A and the received certificate to the certificate log maintainer. The time t_A is used to prevent replay attacks, and will later be used for accountability. The certificate log maintainer checks whether t_A is in an acceptable time range (e.g. t_A is in the same day as his local time). If it is, then he locates the corresponding (rgx, dg^{id}) in dg^{rgx} in the latest record of his log such that *example.com* is an instance of regular expression rgx, locates $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id} and $cert$ in dg^a, then generates the proof of presence of $cert$ in dg^a, $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id}, (rgx, dg^{id}) in dg^{rgx}, and $\mathsf{h}(req, N, dg^{rgx})$ is the latest record in the digest dg_{clog} of the log with size N. Then, the certificate log maintainer signs (dg_{clog}, N, t_A, h), where $h = \mathsf{h}(dg^a, dg^{rv}, rgx, dg^{id}, dg^{rgx}, P)$, and P is the set of proofs, and sends $(dg_{clog}, N, dg^a, dg^{rv}, rgx, dg^{id}, dg^{rgx}, \sigma, P)$ to Alice.

After verifying the signature and proofs, Alice sends the previously stored dg'_{clog} with the size N' to the log maintainer, and expects to receive the proof of extension of (dg'_{clog}, N') into (dg_{clog}, N). If they all valid, then Alice replaces the corresponding cache by the signed (dg_{clog}, N, t_A, h) and believes that the certificate is an authentic one.

In order to preserve privacy of Alice's browsing history, instead of asking Alice to query all proofs from the log maintainer, Alice can send the request to Bob who will redirect the request to the log maintainer, and redirect the received proofs from the log maintainer to Alice.

With DTKI, Alice is able to verify whether Bob's domain has a certificate by querying the proof of absence of certificates for *example.com* in the corresponding certificate log. This is useful to prevent TLS stripping attacks, where an attacker can maliciously convert a HTTPS connection into a HTTP connection.

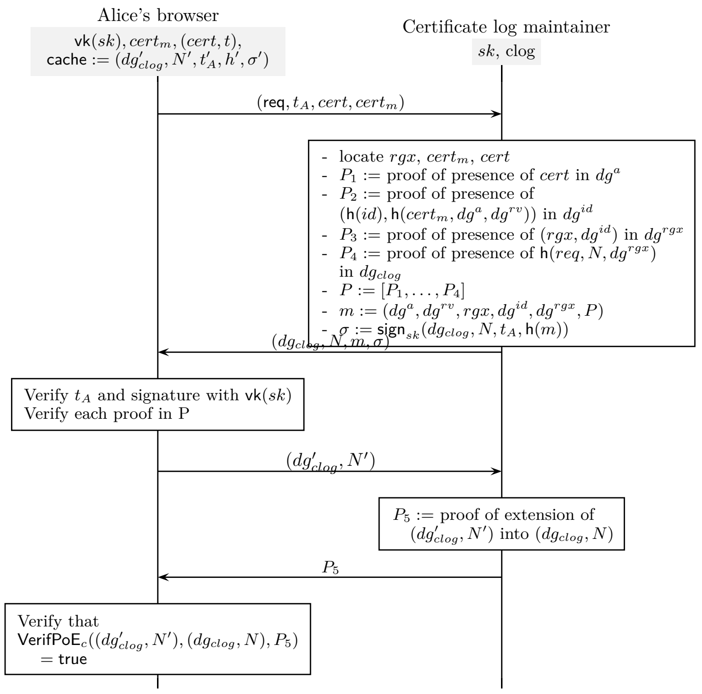

Alice's browser
$\mathsf{vk}(sk), cert_m, (cert, t)$,
$\mathsf{cache} := (dg'_{clog}, N', t'_A, h', \sigma')$
Certificate log maintainer
sk, clog
$(\mathsf{req}, t_A, cert, cert_m)$
- locate rgx, $cert_m$, $cert$
- $P_1 :=$ proof of presence of $cert$ in dg^a
- $P_2 :=$ proof of presence of $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a, dg^{rv}))$ in dg^{id}
- $P_3 :=$ proof of presence of (rgx, dg^{id}) in dg^{rgx}
- $P_4 :=$ proof of presence of $\mathsf{h}(req, N, dg^{rgx})$ in dg_{clog}
- $P := [P_1, \ldots, P_4]$
- $m := (dg^a, dg^{rv}, rgx, dg^{id}, dg^{rgx}, P)$
- $\sigma := \mathsf{sign}_{sk}(dg_{clog}, N, t_A, \mathsf{h}(m))$
$(dg_{clog}, N, m, \sigma)$
Verify t_A and signature with $\mathsf{vk}(sk)$
Verify each proof in P
(dg'_{clog}, N')
$P_5 :=$ proof of extension of (dg'_{clog}, N') into (dg_{clog}, N)
P_5
Verify that $\mathsf{VerifPoE}_c((dg'_{clog}, N'), (dg_{clog}, N), P_5) = \mathsf{true}$

Fig. 3. The protocol for verifying a certificate with the corresponding certificate log maintainer.

4.3. Log verification

To verify whether a certificate log authorised for Bob's domain contains fake certificates, Bob needs to periodically check that all certificates for his domain recorded in the certificate log are authentic. To do so, he can check all certificates for his domain stored in the certificate log, and verify the proof that the corresponding digest (i.e. dg^a and dg^{rv}) are recorded in the certificate log. Note that every time when a certificate log maintainer is blacklisted by the mapping log maintainer, Bob needs to run this verification to check his certificates.

In addition, we need to ensure that the mapping log maintainer and certificate log maintainers behaved honestly. In particular, we need to ensure that the mapping log maintainer and certificate log maintainers did update their log correctly according to the request, and certificate log maintainers did follow the latest mappings specified in the mapping log.

These checks can be easily done if there are trusted third parties (TTPs) who can monitor the log. However, since we aim to provide a TTP-free system, DTKI uses a crowdsourcing-like method, based on random checking, to monitor the correctness of the public log. The basic idea of random checking is that each user randomly selects a record in the log, and verifies whether the request and data in this record have been correctly managed. If all records are verified, the entire log is verified. Users only need to run the random checking periodically (e.g. once a day). The full version (with formalisation) of random checking can be found in our technical report. We give a flavour here by providing some examples. Example 4.1 presents the random checking process to verify the correct behaviour of the mapping log.

Example 4.1. If the verifier has randomly selected the k^{th} record labelled by $\mathsf{h}(\mathsf{add}(rgx, id), t_k, dg^s_k, dg^{bl}_k, dg^r_k, dg^i_k)$ in the mapping log, then it means that all digests in this record are updated from the $(k-1)^{\text{th}}$ record by adding a new mapping (rgx, id) in the mapping log at time t_k.

Let the label of the $(k-1)^{\text{th}}$ record be $\mathsf{h}(req_{k-1}, t_{k-1}, dg^s_{k-1}, dg^{bl}_{k-1}, dg^r_{k-1}, dg^i_{k-1})$, then to verify the correctness of this record, the verifier should run the following process:

— verify that $dg^s_k = dg^s_{k-1}$ and $dg^{bl}_k = dg^{bl}_{k-1}$; and
— verify that dg^r_k is the result of adding (rgx, id) into dg^r_{k-1} by using $\mathsf{VerifPoAdd}_o$, and id is an instance of rgx; and
— verify that (id, dg^{irgx}_k) is the result of replacing (id, dg^{irgx}_{k-1}) in dg^i_{k-1} by (id, dg^{irgx}_k) by using $\mathsf{VerifPoM}_o$; and
— verify that dg^{irgx}_k is the result of adding rgx into dg^{irgx}_{k-1} by using $\mathsf{VerifPoAdd}_o$.

Note the all proofs required in the above are given by the log maintainer. If the above tests succeed, then the mapping log maintainer has behaved correctly for this record. $\diamond$

The verification on the certificate log is similar to the mapping log. However, there is one more thing needed to be verified – the synchronisation between the mapping log and certificate logs. This verification includes that the certificate log only manage the certificates for domains they are authorised to (according to the mapping log); and if there are modifications on the mapping, then the corresponding certificate log maintainer should add or remove all certificates according to the modified mapping. We present an example to show what a verifier should do to verify that the certificate log was authorised to add or remove a certificate.

Example 4.2. If the verifier has randomly selected the k^{th} record labelled by $\mathsf{h}(\mathsf{reg}(\mathsf{sign}_{sk}(cert_{TLS}, t)), N_k, dg^{rgx}_k)$ in the certificate log, where dg^{rgx}_k is the digest of ordered sequence of format (rgx, dg^{id}_k), dg^{id}_k is the digest of ordered sequence of format $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a_k, dg^{rv}_k))$, $cert_m$ is a master certificate, and $cert_{TLS}$ is a TLS certificate. Let dg^{rgx}_k be the digest dg^{rgx} in the $k-1^{\text{th}}$ record, and similarly for dg^{id}_{k-1}, dg^a_{k-1} ,dg^{rv}_{k-1}. Let the subject of $cert_{TLS}$ be id'. The verifier should verify the following tests:

— Verify that $\mathsf{sign}_{sk}(cert_{TLS}, t)$ is correctly signed according to $cert_m$; and
— Verify that $cert_m$ is not expired, and shares the same subject id' with $cert_{TLS}$, and $id' = id$; and
— Verify that dg^a_k is the result of adding $cert_{TLS}$ into dg^a_{k-1}; and
— Verify that dg^{id}_k is the result of replacing
$(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a_{k-1}, dg^{rv}_{k-1}))$ by $(\mathsf{h}(id), \mathsf{h}(cert_m, dg^a_k, dg^{rv}_k))$ in dg^{id}_{k-1}; and
— Verify that $dg^{rv}_k = dg^{rv}_{k-1}$; and

— Verify that dg_k^{rgx} is the result of replacing (rgx, dg_{k-1}^{id}) by (rgx, dg_k^{id}) in dg_{k-1}^{rgx}; and
— Verify that (rgx', id'') is in the $dg_{N_k}^r$ in the N_k^{th} element of the mapping log, such that $rgx' = rgx$, and id' is the identity of the certificate log.

If the above tests succeed, then the certificate log maintainer behaves correctly on this record. $\diamond$

4.4. Performance Evaluation

In this section, we measure the cost of different protocols in DTKI.

Assumptions. We assume that the size of a certificate log is 10^8 (the total number of registered domain names currently is 2.71×10^8 [Num 2014], though only a fraction of them have certificates). In addition, we assume that the number of stored regular expressions, the number of certificate logs, and the size of the mapping log are 1000 each. (In fact, if we assume a different number or size (e.g. 100 or 10000) for them, it makes almost no difference to the conclusion). Moreover, in the certificate log, we assume that the size of the set of data represented by dg^{rgx} is 10, by dg^{id} is 10^5, by dg^a is 10, and by dg^{rv} is 100. These assumptions are based on the fact that dg^{rgx} represents the set of regular expressions maintained by a certificate log; the dg^{id} represents the set of domains which is an instance of a regular expression; and dg^a and dg^{rv} represent the set of currently valid certificates and the revoked certificates, respectively. Furthermore, we assume that the size of a certificate is 1.5 KB, the size of a signature is 256 bytes, the length of a regular expression and an identity is 20 bytes each, and the size of a digest is 32 bytes.

Space. Based on these assumptions, the approximate size of the transmitted data in the protocol for publishing a certificate is 4 KB, for requesting a mapping is 3 KB, and for verifying a certificate is 5 KB. Since the protocols for publishing a certificate and requesting a mapping are run occasionally, we mainly focus on the cost of the protocol for verifying a certificate, which is required to be run between a log server and a web browser in each secure connection.

By using Wireshark, we[1] measure that the size of data for establishing an HTTPS protocol to login to the internet bank of HSBC, Bank of America, and Citibank are 647.1 KB, 419.9 KB, and 697.5 KB, respectively. If we consider the average size ($\approx$588 KB) of data for these three HTTPS connections, and the average size ($\approx$6 KB) of date for their corresponding TLS establishment connections, we have that in each connection, DTKI incurs 83% overhead on the cost of the TLS protocol. However, since the total overhead of a HTTPS connection is around 588 KB, so the cost of DTKI only adds 0.9% overhead to each HTTPS connection, which we consider acceptable.

Time. Our implementation uses a SHA-256 hash value as the digest of a log and a 2048 bit RSA signature scheme. The time to compute a hash[2] is $\approx$ 0.01 millisecond (ms) per 1KB of input, and the time to verify a 2048 bit RSA signature is 0.48 ms. The approximate verification time on the user side needed in the protocol for verifying certificates is 0.5 ms.

Hence, on the user side, the computational cost on the protocol for verifying certificates incurs 83% on the size of data for establishing a TLS protocol, and 9% on the size of data for establishing an HTTPS protocol; the verification time on the protocol for verifying certificates is 1.25 % of the time for establishing a TLS session (which is approximately 40 ms measured with Wireshark on the TLS connection to HSBC bank).

[1]We use the MacBook Air 1.8 GHz Intel Core i5, 8 GB 1600 MHz DDR3.
[2]SHA-256 on 64 byte size block.

5. SECURITY ANALYSIS

We consider an adversary who can compromise the private key of all infrastructure servers in DTKI. In other words, the adversary can collude with all log servers and certificate authorities to launch attacks.

Main result. Our security analysis shows that

— if the distributed random checking has verified all required tests, and domain owners have successfully verified their initial master certificates, then DTKI can prevent attacks from the adversary; and

— if the distributed random checking has not completed all required tests, or domain owners have not successfully verified their initial master certificates, then an adversary can launch attacks, but the attacks will be detected afterwards.

We provide all source codes and files required to understand and reproduce our security analysis at [Yu 2015]. In particular, these include the complete DTKI models and the verified proofs.

5.1. Formal analysis

We analyse the main security properties of the DTKI protocol using the TAMARIN prover [Meier et al. 2013]. The TAMARIN prover is a symbolic analysis tool that can prove properties of security protocols for an unbounded number of instances and supports reasoning about protocols with mutable global state, which makes it suitable for our log-based protocol. Protocols are specified using multiset rewriting rules, and properties are expressed in a guarded fragment of first order logic that allows quantification over timepoints.

TAMARIN is capable of automatic verification in many cases, and it also supports interactive verification by manual traversal of the proof tree. If the tool terminates without finding a proof, it returns a counter-example. Counter-examples are given as so-called dependency graphs, which are partially ordered sets of rule instances that represent a set of executions that violate the property. Counter-examples can be used to refine the model, and give feedback to the implementer and designer.

Modeling aspects. We used several abstractions during modeling. We model our log as lists, similar to the abstraction used in [Basin et al. 2014; Yu et al. 2015]. We also assume that the random checking is verified.

We model the protocol roles D (domain server), M (mapping log maintainer), C (certificate log maintainer), and CA (certificate authority) by a set of rewrite rules. Each rewrite rule typically models receiving a message, taking an appropriate action, and sending a response message. Our modeling approach is similar to the one used in most TAMARIN models. Our modeling of the roles directly corresponds to the protocol descriptions in the previous sections. TAMARIN provides built-in support for a Dolev-Yao style network attacker, i.e., one who is in full control of the network. We additionally specify rules that enable the attacker to compromise service providers, namely teh mapping log maintainer, certificate log maintainers and CAs, learn their secrets, and modify public logs.

The final DTKI model consists of 959 lines for the base model and five main property specifications, examples of which we will give below.

Proof goals. We state several proof goals for our model, exactly as specified in TAMARIN's syntax. Since TAMARIN's property specification language is a fragment of first-order logic, it contains logical connectives (`|`, `&`, `==>`, `not`, ...) and quantifiers (`All`, `Ex`). In TAMARIN, proof goals are marked as `lemma`. The `#`-prefix is used to denote timepoints, and "`E @ #i`" expresses that the event E occurs at timepoint i.

The first goal is a check for executability that ensures that our model allows for the successful transmission of a message. It is encoded in the following way.

```
lemma protocol_correctness:
 exists-trace
 " /* It is possible that */
   Ex D Did n rgx ltpkD stpkD #i1.

   /* The user received a confirmation of receiving the message
   i.e. hashed secret the user has sent to the domain */

   Com_Done(D, Did, n, rgx, ltpkD, stpkD) @ #i1

     /* without the adversary compromising any party. */
   &   not (Ex #i2 CA ltkCA.
             Compromise_CA(CA,ltkCA) @ #i2)

   &   not (Ex #i3 C ltkC.
             Compromise_CLM(C,ltkC) @ #i3)

   &   not (Ex #i4 M ltkM.
             Compromise_MLM(M,ltkM) @ #i4)

"
```

The property holds if the TAMARIN model exhibits a behaviour in which a domain server received a message without the attacker compromising any service providers. This property mainly serves as a sanity check on the model. If it did not hold, it would mean our model does not model the normal (honest) message flow, which could indicate a flaw in the model. TAMARIN automatically proves this property in several minutes and generates the expected trace in the form of a graphical representation of the rule instantiations and the message flow.

We additionally proved several other sanity-checking properties to minimize the risk of modeling errors.

The second example goal is a secrecy property with respect to a classical attacker, and expresses that when no service provider is compromised, the attacker cannot learn the message exchanged between a user and a domain server. Note that `K(m)` is a special event that denotes that the attacker knows m at this time.

```
lemma message_secrecy_no_compromised_party:
 "
 All D Did m rgx ltpkD stpkD #i1.

   /* The user received a confirmation of receiving the message
   i.e. hashed secret the user has sent to the domain */

   (Com_Done(D, Did, m, rgx, ltpkD, stpkD) @ #i1

/* and no party has been compromised */
   &   not (Ex #i2 CA ltkCA.
             Compromise_CA(CA,ltkCA) @ #i2)

   &   not (Ex #i3 C ltkC.
             Compromise_CLM(C,ltkC) @ #i3)

   &   not (Ex #i4 M ltkM.
             Compromise_MLM(M,ltkM) @ #i4)
      )
      ==>
      ( /* then the adversary cannot know m */
       not (Ex #i5. K(m) @ #i5)
      )
 "
```

TAMARIN proves this property automatically (in 575 steps).

The above result implies that if a domain server D, whose domain name is Did such that Did is an instence of regular expression rgx, receives a message that was sent by a user, and the attacker did not compromise server providers, then the attacker will not learn the message.

The next two properties encode the unique security guarantees provided by our protocol, in the case that even all service providers are compromised.

The first main property we prove is that when all service providers (i.e. CAs, the MLM, and CLMs) are compromised, and the domain owner has successfully verified his master certificate in the log, then the attacker cannot learn the message exchanged between a user and a domain owner. It is proven automatically by TAMARIN in 5369 steps.

```
lemma message_secrecy_compromise_all_domain_verified_master_cert:
 "
 All D Did m rgx ltpkD stpkD #i1.

   /* The user received a confirmation of receiving the message
   i.e. hashed secret the user has sent to the domain */

   (Com_Done(D, Did, m, rgx, ltpkD, stpkD) @ #i1

   /* and at an earlier time, the domain server has verified his
      master certificate */
     & Ex #i2.
     VerifiedMasterCert(D, Did, rgx, ltpkD) @ #i2
     & #i2 < #i1

/* and all parties can be compromised*/

      )
      ==>
      ( /* then the adversary cannot know m */
       not (Ex #i3. K(m) @ #i3)
      )
 "
```

The property states that if a domain server D receives a message that was sent by a user, and at an earlier time, the domain server has verified his master certificate, then even if the attacker can compromise all server providers, the attacker cannot learn the message.

The final property states that when all service providers can be compromised, and a domain owner has not verified his/her master certificate, and the attacker learns the message exchanged between a user and the domain owner, then afterwards the domain owner can detect this attack by checking the log. It is also verified by TAMARIN within a few minutes.

```
lemma detect_bad_records_in_the_log_when_master_cert_not_verified:

 "
 All D Did m rgx ltpkD flag stpkD #i1 #i2 #i3.

   /* The user received a confirmation of receiving the message
   i.e. hashed secret the user has sent to the domain */

   (Com_Done(D, Did, m, rgx, ltpkD, stpkD) @ #i1

   /* and all parties can be compromised*/

   /* and the master certificate of the domain was not
      initially verified */

   /* and the adversary knows m */

     &  K(m) @ #i2

   /* and we afterwards check the log */
     &  CheckedLog(D, Did, rgx, ltpkD, flag, stpkD) @ #i3
     &   #i1 < #i3)
      ==>
     ( /* then we can detect a fake record in the log */
        (flag = 'bad')
     )
 "
```

6. COMPARISON

	SK [Eckersley 2012]	CT [Laurie et al. 2013]	AKI [Kim et al. 2013]	ARPKI [Basin et al. 2014]	DTKI
Terminology					
Log provider	Time-line server	Log	Integrity log server (ILS)	Integrity log server (ILS)	Certificate/Mapping log maintainer (CLM, MLM)
Log extension	-	Log consistency	-	-	Log extension
Trusted party	Mirror	Auditor & monitor	Validator	Validator (optional)	-
Whether answers to queries rely on trusted parties or are accompanied by a proof					
Certificate-in-log query:	Rely	Proof	Proof	Proof	Proof
Certificate-current-in-log query:	Rely	Rely	Proof	Proof	Proof
Subject-absent-from-log query:	Rely	Rely	Proof	Proof	Proof
Log extension query:	Rely	Proof	Rely	Rely	Proof
Non-necessity of trusted party					
Trusted party role can be distributed randomly to browsers	No	No	No+	No+	Yes
Trust assumptions					
Not all service providers collude together	Yes	Yes	Yes	Yes	No
Domain is initially registered by an honest party	No	No	Yes*	Yes*	Yes*
Security guarantee					
Attacks detection or prevention	Detection	Detection	Prevention	Prevention	Prevention
Oligopoly issues					
Log providers required to be built into browser (oligopoly)	Yes	Yes	Yes	Yes	Only MLM
Monitors required to be built into browser (oligopoly and trust non-agility)	Yes	No	Yes	Yes†	No

+ The system limits the trust in each server by letting them to monitor each other's behaviour.
* Without the assumption, the security guarantee is detection rather than prevention.
† The trusted party is optional, if there is a trusted party, then the trusted party is required to be built into browser.

Fig. 4. Comparison of log-based approaches to certificate management. **Terminology** helps compare the terminology used in the papers. **How queries rely on trusted parties** shows whether responses to browser queries come with proof of correctness or rely on the honesty of trusted parties. **Necessity of trusted parties** shows whether the TP role can be performed by browsers. **Trust assumptions** shows the assumption for the claimed security guarantee. **Oligopoly issues** shows the entities that browsers need to know about.

As mentioned previously, DTKI builds upon a wealth of ideas from SK [Eckersley 2012], CT [Laurie et al. 2013], CIRT [Ryan 2014], and AKI [Kim et al. 2013]. Figure 4 shows the dimensions along which DTKI aims to improve on those systems.

Compared with CT, DTKI supports revocation by enabling log providers to offer proofs of *absence* and *currency* of certificates. In CT, there is no mechanism for revocation. CT has proposed additional data structures to hold revoked certificates, and those data structures support proofs of their contents. However, there is no mechanism to ensure that the data structures are maintained correctly in time.

Compared to CIRT, DTKI extends the log structure of CIRT to make it suitable for multiple log maintainers, and provides a stronger security guarantee as it prevents attacks rather than merely detecting them. In addition, the presence of the mapping log maintainer and multiple certificate log maintainers create some extra monitoring work. DTKI solves it by using a detailed crowd-sourcing verification system to distribute the monitoring work to all users' browsers.

Compared to AKI and ARPKI, in DTKI the log providers can give proof that the log is maintained append-only from one step to the next. The data structure in A(RP)KI does not allow this, and therefore they cannot give a verifiable guarantee to the clients that no data is removed from the log.

DTKI improves the support that CT and A(RP)KI have for multiple log providers. In CT and AKI, domain owners wishing to check if there exists a log provider that has registered a certificate for him has to check all the log providers, and therefore the full set of log providers has to be fixed and well-known. This prevents new log providers being flexibly created, creating an oligopoly. In contrast, DTKI requires the browsers only to have the MLM public key built-in, minimising the oligopoly element.

In DTKI, no trusted party is required, as it uses crowd-sourcing verification to eliminate the need of trusted parties, i.e. a trusted party's verification work can be done probabilistically in small pieces, meaning that users' browsers can collectively perform the monitoring role.

Unlike the mentioned previous work, DTKI allows the possibility that all service providers (i.e. the MLM, CLMs, and mirrors) to collude together, and can still prevent attacks. In contrast, SK and CT can only detect attacks, and to prevent attacks, A(RP)KI requires that not all service providers collude together. Similar to A(RP)KI, DTKI also assumes that the domain is initially registered by an honest party to prevent attacks, otherwise A(RP)KI and DTKI can only detect attacks.

7. DISCUSSION

Coverage of random checking. As mentioned, several aspects of the logs are verified by user's browsers performing randomly-chosen checks. The number of things to be checked depends on the size of the mapping log and certificate logs. The size of the mapping log mainly depends on the number of certificate logs and the mapping from regular expressions to certificate logs; and the size of certificate logs mainly depends on the number of domain servers that have a TLS certificate. Currently, there are 2.71×10^8 domains [Num 2014] (though not every domain has a certificate), and 3×10^9 internet users [Int 2014]. Thus, if every user makes one random check per day, then everything will on average, be checked 10 times per day.

Gossip protocol. As mentioned in the overview, to avoid victims being trapped in a "bubble" created by very powerful attackers who controls the network and all service infrastructures such as ISPs and log maintainers, DTKI assumes the existence of a gossip protocol [Jelasity et al. 2007] that can be used for users to detect if a log maintainer shows different versions (i.e. different pairs of digest and size) of the log to different sets of users. The gossip protocol allows client browsers to exchange with other users the digest and size of the log that they have received in the DTKI protocols. The gossip protocol provides a means for a browser to identify peers with whom to exchange digests. The mobility of phones and laptops help ensure maximum gossip performance. At any time, a user can request a proof that the pair of digest and size currently offered by the log is an extension of a previous pair of digest and size of the log received from other users via the gossip protocol.

Accountability of mis-behaving parties. The main goal of new certificate management schemes such as CT, AKI and DTKI is to address the problem of mis-issued certificates, and to make the mis-behaving (trusted) parties accountable.

In DTKI, a domain owner can readily check for rogue certificates for his domain. First, he queries a mirror of the mapping log maintainer to find which certificate log maintainers (CLM) are allowed to log certificates for the domain (section 4). Then he examines the certificates for his domain that have been recorded by those CLMs. The responses he obtains from the mirror and the CLMs are accompanied by proofs. If he detects a mis-issued certificate, he requests revocation in the CLM. If that is refused, he can complain to the top-level domain, who in turn can request MLM to change the CLM for his domain (after that, the offending CLM will no longer be consulted by browsers). This request can't be refused because MLM is governed by an international panel. The intervening step, of complaining to the top-level domain, reflects the way domain names are actually managed in practice. Different Top-level domains have different terms and conditions, and domain owners take them into account when purchasing domain names. In DTKI, log maintainers are held accountable because they sign and timestamp their outputs. If a certificate log maintainer issues inconsistent digest, this fact will be detected and the log maintainer can be blamed and blacklisted. If the mapping log misbehaved, then its governing panel must meet and resolve the situation.

In certificate transparency, this process is not as smooth. Firstly, the domain owner doesn't get proof that the list of issued certificates is complete; he needs to rely on monitors and auditors. Next, the process for raising complaints with log maintainers who refuse revocation requests is less clear (indeed, the RFC [Laurie et al. 2013] says that what domain owners should do if they see an incorrect log entry is beyond scope of their document). In CT, a domain owner has no ability to dissociate himself from a log maintainer and use a different one.

AKI addresses this problem by saying that log maintainer that refuses to unregister an entry will eventually lose credibility through a process managed by validators, and will be subsequently ignored. The details of this credibility management are not very clear, but it does not seem to offer an easy way for domain owners to control which log maintainers are relied on for their domain.

Avoidance of monopoly. As we mentioned in the introduction, the predecessors (SK, AKI, E(CT)) of DTKI do not solve a foundational issue, namely *monopoly* (or *oligopoly*). These proposals require that all browser vendors agree on a fixed list of log maintainers and/or validators, and build it into their browsers. This means there will be a large barrier to create a new log maintainer.

CT has some support for multiple logs, but it doesn't have any method to allocate different domains to different logs. In CT, when a domain owner wants to check whether misissued certificates are recorded in logs, he needs to contact all existing logs, and download all certificates in each of the logs, because there is no way to prove to the domain owner that no certificates for his domain is in the log, or to prove that the log maintainer has showed all certificates in the log for his domain to him. Thus, to be able to detect fake certificates, CT has to keep a very small number of log maintainers. This prevents new log providers being flexibly created, creating an oligopoly.

In contrast to its predecessors, DTKI does not have a fixed set of certificate log maintainers (CLMs) to manage certificates for domain owners, and it is easy to add or remove a certificate log maintainer by updating the mapping log. DTKI only has one lightweight governing party, i.e. the mapping log maintainer (MLM), which needs to be built into browsers. However, we minimise the monopoly on the MLM (it is hard to be avoid), because

— the MLM has no bias on certain countries since it is maintained by an international panel; and
— the MLM modifies the mapping log only for strategic and long term reasons; it only periodically (e.g. every day) publishes a *signed Mlog timestamp*; and it is not involved in day-to-day management (which is the work of CLMs and mirrors of the mapping log); and
— the MLM is not required to be trusted by users' browsers.

8. CONCLUSIONS AND FUTURE WORK

Sovereign keys (SK), certificate transparency (CT), accountable key infrastructure (AKI), and certificate issuance and revocation transparency (CIRT) are recent proposals to make public key certificate authorities more transparent and verifiable, by using public logs. CT is currently being implemented in servers and browsers. Google is building a certificate transparency log containing all the current known certificates, and is integrating verification of proofs from the log into the Chrome web browser.

Unfortunately, as it currently stands, CT risks creating a monopoly or small oligopoly of log maintainers (as discussed in section 7), of which Google itself will be a principal one. Therefore, adoption of CT risks investing more power about the way the internet is run in a company that arguable already has too much power.

In this paper we proposed DTKI – a TTP-free public key validation system using an improved construction of public logs. DTKI can prevent attacks based on mis-issued certificates, and minimises undesirable oligopoly situations by using the mapping log. In addition, we formalised the public log structure and its implementation; such formalisation work was missing in the previous systems (i.e. SK, CT, AKI, and CIRT). Since devising new security protocols is notoriously error-prone, we provide a formalisation of DTKI, and correctness proofs.

References

Certificate Patrol. (????). http://patrol.psyced.org

The EFF SSL Observatory. (????). https://www.eff.org/observatory

MS01-017: Erroneous VeriSign-Issued Digital Certificates Pose Spoofing Hazard. Microsoft Support. (????). http://support.microsoft.com/kb/293818

2012. Black Tulip Report of the investigation into the DigiNotar Certificate Authority breach. Fox-IT (Tech. Report). (2012).

2014. Domain Name Industry Brief. Verisign. (2014). http://www.verisigninc.com/en_US/innovation/dnib/index.xhtml

2014. Internet Users. Internet Usage Statistics. (2014). http://www.internetlivestats.com/internet-users/

Mansoor Alicherry and Angelos D. Keromytis. 2009. DoubleCheck: Multi-path verification against man-in-the-middle attacks. In *ISCC*. 557–563.

B. Amann, M. Vallentin, S. Hall, and R. Sommer. 2012. Revisiting SSL: A large-scale study of the internet's most trusted protocol. Technical report, ICSI. (2012).

J. Appelbaum. 2011. Detecting Certificate Authority compromises and web browser collusion. Tor Blog. (2011).

C. Arthur. 2011. Rogue web certificate could have been used to attack Iran dissidents. The Guardian. (2011).

David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. 2014. ARPKI: Attack Resilient Public-key Infrastructure.. In *ACM Conference on Computer and Communications Security.*

Jeremy Clark and Paul C. van Oorschot. 2013. SSL and HTTPS:Revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE Symposium on Security and Privacy.*

T. Dierks and E. Rescorla. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard). (Aug. 2008). http://www.ietf.org/rfc/rfc5246.txt Updated by RFCs 5746, 5878, 6176.

Peter Eckersley. 2011. Iranian hackers obtain fraudulent HTTPS certificates: How close to a Web security meltdown did we get? Electronic Frontier Foundation. (2011). https://www.eff.org/deeplinks/2011/03/iranian-hackers-obtain-fraudulent-https

P. Eckersley. 2012. Sovereign key cryptography for internet domains. Internet Draft. (2012).

Peter Eckersley and Jesse Burns. Is the SSLiverse a safe place? Chaos Communication Congress, 2010. (????).

N. Falliere, L. O. Murchu, and E. Chien. 2011. W32.Stuxnet Dossier. Technical report, Symantec Corporation. (2011).

Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. 2007. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)* 25, 3 (2007), 8.

Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. 2013. Accountable Key Infrastructure (AKI): A Proposal for a Public-Key Validation Infrastructure. In *the 22nd International World Wide Web Conference (WWW 2013).*

A. Langley. 2011. Public-key pinning. *ImperialViolet* (blog). (2011).

A. Langley. 2012. Revocation checking and Chrome's CRL. *ImperialViolet* (blog). (2012).

Ben Laurie and Emilia Kasper. 2012. Revocation Transparency. (September 2012). Google Research.

B. Laurie, A. Langley, and E. Kasper. 2013. Certificate Transparency. RFC 6962 (Experimental). (2013).

John Leyden. 2012. Trustwave admits crafting SSL snooping certificate: Allowing bosses to spy on staff was wrong, says security biz. The Register. (2012). www.theregister.co.uk/2012/02/09/tustwave_disavows_mitm_digital_cert

M. Marlinspike. 2011. SSL and the future of authenticity. In *Black Hat, USA*.

M. Marlinspike and T. Perrin. 2012. Trust assertions for certificate keys (TACK). Internet draft. (2012).

Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification, 25th International Conference, CAV 2013, Princeton, USA, Proc. (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, 696–701. DOI:http://dx.doi.org/10.1007/978-3-642-39799-8_48

Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *CRYPTO*. 369–378.

André Niemann and Jacqueline Brendel. 2013. A Survey on CA Compromises. (2013).

Ronald L Rivest. 1998. Can we eliminate certificate revocation lists?. In *Financial Cryptography*. Springer, 178–183.

Paul Roberts. 2011. Phony SSL Certificates issued for Google, Yahoo, Skype, Others. Threat Post. (March 2011). http://threatpost.com/phony-ssl-certificates-issued-google-yahoo-skype.-others-032311

Mark Ryan. 2014. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS*.

Christopher Soghoian and Sid Stamm. 2011. Certified Lies: Detecting and Defeating Government Interception Attacks against SSL. In *Financial Cryptography*. 250–259.

Toby Sterling. 2011. Second firm warns of concern after Dutch hack. Yahoo! News. (September 2011). http://news.yahoo.com/second-firm-warns-concern-dutch-hack-215940770.html

S. Turner and T. Polk. 2011. Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176 (Proposed Standard). (March 2011). http://www.ietf.org/rfc/rfc6176.txt

Dan Wendlandt, David G. Andersen, and Adrian Perrig. 2008. *Perspectives:* Improving SSH-style Host Authentication with Multi-Path Probing. In *USENIX Annual Technical Conference*. 321–334.

Jiangshan Yu. 2015. Tamarin models for the DTKI protocol. (2015). http://www.jiangshanyu.com/doc/paper/DTKI-proof.zip.

Jiangshan Yu, Mark Ryan, and Cas Cremers. 2015. How to detect unauthorised usage of a key. *IACR Cryptology ePrint Archive* 2015 (2015), 486. http://eprint.iacr.org/2015/486

APPENDIX

A. ATTACK ON CIRT

As we mentioned in the introduction, certificate issuance and revocation transparency (CIRT) also uses public log to store the certificates. In fact, they also rely on a chronological data structure and a weaker version of an ordered data structure, in particular there is no proof of addition nor deletion available. However, since the attack is due to the format of each entry, we will assume that CIRT uses our version of ordered data structure for simplicity's sake.

In CIRT, the log is a chronological data structure over a set of element of the form $(cert, dg)$ where:

— $cert$ is a certificate; and
— dg is the digest of an ordered data structure storing element of the form $(subj, (cert_1, cert_2, \ldots, cert_N))$ where N is fixed and for all $i \in \{1, \ldots, N\}$, $cert_i$ can either be null or a certificate whose identifier is $subj$.

In this system, there can only be one certificate per subject. Hence, in an element $(subj, (cert_1, cert_2, \ldots, cert_N))$, the certificate $cert_1$ represents the active certificate and $cert_2, \ldots, cert_N$ represent the revoked ones. Let us now detail how the log is updated when a new certificate $cert_0$ is sent for $subj_0$. Assume that $(cert_\ell, dg_\ell)$ is the last entry of the log. To be updated, the log will add the entry $(cert_0, dg)$ to the chronological data structure where dg is updated from dg_ℓ as follows: Let us denote D the ordered data structure whose digest is dg_ℓ.

— either for all element $(subj, (cert_1, cert_2, \ldots, cert_N)) \in D$, $subj \neq subj_0$ and so dg is the digest of a structure D' storing the elements of D plus $(subj_0, (cert_0, null, \ldots, null))$
— to there exists an element $(subj_0, (cert_1, cert_2, \ldots, cert_N)) \in D$ and so dg is the digest of a structure D' storing the elements of D where $(subj_0, (cert_1, cert_2, \ldots, cert_N))$ is replaced by $(subj_0, (cert_0, cert_1, \ldots, cert_{N-1}))$.

By keeping the revoked certificates, the authors of CIRT claims that a subject $subj$ can efficiently check the certificates that were used for him by requesting (with the associated proofs) the element $(subj, (cert_1, cert_2, \ldots, cert_N))$ stored in the data structure represented by the digest of the last entry in the chronological data structure, hence it requires only two proofs of presence of an element in a data structure.

The authors fixed N to avoid having to consider unbounded list for efficiency reason. However, fixing N lead to the following attack: Consider a malicious log maintainer that want to add a fake certificate $cert_a$ for $subj_0$ while still managing to be undetected by the tests performed by users. For sake of simplicity, we will describe the different states of the log by only representing the entry corresponding to $subj_0$ in the latest added ordered data structure in the log:

— before the attack, let us assume that the log is storing $(subj_0, (cert_1, cert_2, \ldots, cert_N))$.
— when the log maintainer launchs his attack, he adds $cert_a$ to his log hence $(subj_0, (cert_a, cert_1, cert_2, \ldots, cert_{N-1}))$ will be stored in the log.
— after the attack, to cover his track, the log maintainer will add successively $cert_N, cert_{N-1}, \ldots, cert_1$ to the log which implies that $(subj_0, (cert_1, cert_2, \ldots, cert_N))$ will be once again stored in the log.

Thus, by only checking the last ordered data structure stored in the log, the user $subj$ cannot notice that a fake certificate $cert_a$ was register in his stead. He can only do that by checking every entry of the chronological data structure which is considered as impractical.

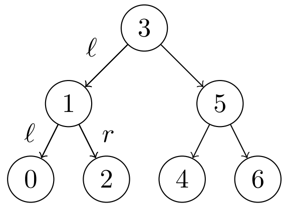


Fig. 5. Total order on the nodes of a tree T=$\{(0, \ell \cdot \ell), \ldots, (6, r \cdot r)\}$

B. IMPLEMENTATION OF CHRONOLOGICAL DATA STRUCTURE

The public logs in our protocol are organised by using two data structures − chronological and ordered data structure. In this section, we show how the chronological data structure can be implemented by the a tree structure called *ChronTree*. We consider the alphabet $\mathcal{A} = \{\ell, r\}$.

Definition B.1. A binary tree T over a set D is a subset of $D \times \mathcal{A}^*$, *i.e.* $T \subset D \times \mathcal{A}^*$, such that

— for all $(d, w), (d', w') \in T$, $w = w'$ implies that $d = d'$; and
— there exists $d \in D$ such that $(d, \varepsilon) \in T$; and
— for all $(d, w) \in T$, if $w = w' \cdot a$ with $a \in \mathcal{A}$ then there exists $d' \in D$ such that $(d', w') \in T$.

The elements of T are called nodes of T. More generally, a *node of* T is an element of $D \times \mathcal{A}^*$; each element of $\mathcal{A}^*$ is called a position and the elements of D are labels. We denote by $\mathsf{pos}(n)$ the position of a node n. A *root* is a node whose position is ε. We will denote by $\mathsf{root}(T)$ the root of the tree T.

We say that a node is the *left child* (resp. *right child*) of a node n in T, denoted $\mathsf{Lchild}_T(n)$ (resp. $\mathsf{Rchild}_T(n)$), if $\mathsf{Lchild}_T(n) \in T$ and $\mathsf{pos}(\mathsf{Lchild}_T(n)) = \mathsf{pos}(n) \cdot \ell$ (resp. $\mathsf{Rchild}_T(n) \in T$ and $\mathsf{pos}(\mathsf{Rchild}_T(n)) = \mathsf{pos}(n) \cdot r$). Furthermore, the parent of a node n in T, denoted $\mathsf{Parent}_T(n)$, is the node $n' \in T$ such that $\mathsf{pos}(n) = \mathsf{pos}(n') \cdot a$ with $a \in \mathcal{A}$. The sibling of a node n in T is the node different from n denoted $\mathsf{Sib}_T(n)$ in T such that $\mathsf{Parent}_T(n) = \mathsf{Parent}_T(\mathsf{Sib}_T(n))$. At last, a node that does not have any child is called a *leaf*.

With these notations, in Definition B.1, the first item indicates that there cannot be two different nodes with the same position in a binary tree; the second item tells us that there must be a root in a tree and finally the last item indicates that each node different from the root must have a parent node.

The path to a node n in T is a sequence of nodes $n_1, \ldots, n_k$ for some k such that $n_1 = \mathsf{root}(T)$, $n_k = n$, and for all $i \in [1, k-1]$, $n_i = \mathsf{Parent}_T(n_{i+1})$. In addition, any node in this sequence is called an *ancestor* of n in T; and a node is called an *descendant* of its ancestors. Moreover, the *depth* of n is $k-1$, denoted $\mathsf{depth}_T(n)$. The *height* of a binary tree T, denoted $\mathsf{height}(T)$, is defined as the maximum of the depths of all nodes in T. We consider that the height of an empty binary tree is -1.

Definition B.2. Given a binary tree T, we define $\mathcal{O}_T$, called in-order tree traversal, as the least transitive relation such that:

— $n\, \mathcal{O}_T\, n'$ if $n \in \mathsf{Lchild}_T(n')$ or $n' \in \mathsf{Rchild}_T(n)$; and
— $\mathcal{O}_T$ is the transitive closure.

An example of the order $\mathcal{O}_T$ in a tree T is given in Figure 5. In which, $0\, \mathcal{O}_T\, 1\, \mathcal{O}_T\, 2\, \mathcal{O}_T\, 3\, \mathcal{O}_T\, 4\, \mathcal{O}_T\, 5\, \mathcal{O}_T\, 6$. A binary tree T is *full* if every non-leaf node in T has exactly two child nodes; it is *complete* when the depth of each leaf is either $\mathsf{height}(T)$ or

$\mathsf{height}(T) - 1$. Finally, we say that T is *perfect* if it is full and all leaves in T have the same depth.

The notion of *Merkle tree* was first introduced in [Merkle 1987] to tackle the issue of secure verification of large amount of data. Typically, a Merkle tree is a binary tree where each non-leaf node is labelled with the hash of the labels of its children; and where each leaf is labeled with some data. Relying on the cryptographic properties of the hash function, this data structure allows one to provide formal proofs of the existence of some data in a Merkle tree; the size of those proofs being logarithmic in the number of data in a complete tree. The small size of proofs makes the usage of Merkle trees particularly interesting when dealing with huge amount of data. Thus, the Certificate Transparency [Laurie et al. 2013] considers a data structure for their public log based on Merkle trees, that we call *ChronTree*.

Since Merkle tree, and so ChronTree, relies on a hash function, we will now consider that all our trees are labeled with bitstrings, *i.e.* sequences of 0 and 1. Moreover, for the rest of the paper, we will consider a hash function denoted h. In practice, the hash function h could be any secure cryptographic hash functions like SHA-256. For the analysis of our system, we will assume that the hash function satisfies the two following properties:

(1) h is injective, that is $\mathsf{h}(a) = \mathsf{h}(b)$ implies that $a = b$.
(2) it is impossible to retrieve an input x from its hash $\mathsf{h}(x)$, *i.e.* there no polynomial probabilistic turing machine h^{-1} such that $\mathsf{h}^{-1}(\mathsf{h}(x)) = x$.

These are classic properties required for cryptographic hash functions and that are satisfied by SHA-256 with overwhelming probability.

Definition B.3. A *ChronTree* T is a full *binary tree* over bitstrings such that:

— for all non-leaf nodes n in T, n is labeled with $\mathsf{h}(t_\ell, t_r)$ where t_ℓ (resp. t_r) is the label of its left child (resp. right child); and
— for all nodes n in T, the subtree of T rooted by the left child of n is perfect and its height is equal or greater than the height of the subtree of T rooted by the right child of n.

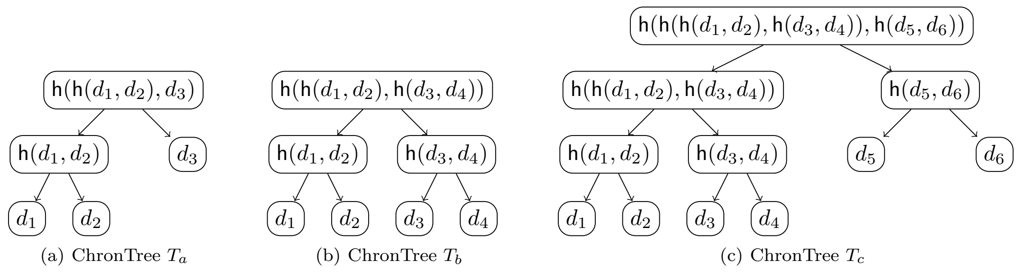


Fig. 6. Examples of ChronTree

Note that a ChronTree is a not necessarily a complete tree. The three trees in Figure 6 are examples of ChronTrees where the datas stored are the bitstrings denoted $d_1, \ldots, d_6$. In fact a ChronTree is completely defined by the sequence of data stored in the leaves as indicated by the next Lemma.

LEMMA B.1. *Consider a sequence of bitstrings $[d_1, \ldots, d_k]$. There exists a unique ChronTree T such that T contains k leaves and if we denote by $n_1, \ldots, n_k$ the leaves of T such that for all $i, j \in \{1, \ldots, k\}$, if $i < j$ then $n_i \mathcal{O}_T n_j$, and for all $i \in \{1, \ldots, k\}$, the leaf n_i is labeled by d_i.*

Proof. We prove it by induction on k and our proof relies on the fact that there exists a unique binary representation of k.

Base case: $k = 1$. We have $k = 2^0$, $S = [d_1]$, and T contains one leaf n_1. Since T is a ChronTree , then $\mathsf{root}(T) = n_1$. If n_1 is labelled by d_1, then T contains only one node (d_1, ε). So the lemma is true when $k = 1$.

Induction case: we assume that the lemma is true when $k \leq n$. We prove that the lemma is also true when $k = n + 1$.

— If $k = 2^{k'}$ for some k', and since T is a ChronTree, we know that the tree is prefect. Thus, the subtree T_{lsub} and T_{rsub} rooted respectively from $\mathsf{Lchild}_T(\mathsf{root}(T))$ and $\mathsf{Rchild}_T(\mathsf{root}(T))$ contains the same number of leaves, i.e. $2^{k'-1}$. Also, by inductive hypothesis, we know that the lemma is true when $k \leq n$, and since $2^{k'-1} < n$, so both T_{lsub} and T_{rsub} are unique. Thus, the ChronTree with $n + 1$ leaves is also unique; otherwise
— Let $k = 2^{a_1} + \ldots + 2^{a_m}$ such that for all $i, j \in [1, m]$, if $i < j$, then $a_i > a_j$. Since T is a ChronTree, then we know that the subtree T_{lsub} rooted from $\mathsf{Lchild}_T(\mathsf{root}(T))$ is perfect according to the Def. B.3. Thus, T_{lsub} contains 2^{a_1} leaves, and T_{lsub} contains $2^{a_2} + \ldots + 2^{a_m}$. By inductive hypothesis, we know that the lemma is true when $k \leq n$, and since $2^{a_1} < n$ and $2^{a_2} + \ldots + 2^{a_m} < n$, we have that both T_{lsub} and T_{rsub} are unique. Thus, the ChronTree with $n + 1$ leaves is also unique.

Thus, by the induction proof, the lemma is true. □

Following Lemma B.1, given a ChronTree T with k leaves, we denote by $\mathcal{S}(T) = [d_1, \ldots, d_k]$ the sequence of bitstrings stored in T. Reciprocally, given a sequence of bitstrings $[d_1, \ldots, d_k]$, we denote by $\mathcal{CT}([d_1, \ldots, d_k])$ the unique ChronTree that stores $[d_1, \ldots, d_k]$. Moreover, we say that the *size* of a ChronTree is the number its leaves.

Example B.1. Coming back to Figure 6, we have that $\mathcal{S}(T_a) = [d_1, d_2, d_3]$, $\mathcal{S}(T_b) = [d_1, d_2, d_3, d_4]$ and $\mathcal{S}(T_c) = [d_1, d_2, d_3, d_4, d_5, d_6]$. ◇

Lemma B.2. *Let T, T' be two ChronTrees of size N. Let's denote $\mathcal{S}(T) = [d_1, \ldots, d_N]$ and $\mathcal{S}(T') = [d'_1, \ldots, d'_N]$. Let's denote by h and h' be the respective hash values of T and T'. We have that*

(1) $\mathsf{height}(T) = \mathsf{height}(T')$*; and*
(2) $h = h'$ *if, and only if, for all* $i \in \{1, \ldots, N\}$, $d_i = d'_i$*; and*
(3) *for all* $i \in \{1, \ldots, N\}$, *for all* $w \in \mathcal{A}^*$, $(d_i, w) \in T$ *if, and only if,* $(d'_i, w) \in T'$.

Proof. We do the proof by induction on N.

Base case $N = 1$: In such a case, since a ChronTree is always a full tree, then T and T' are both reduced to their roots, *i.e.* $T = \{(h, \varepsilon)\}$ and $T' = \{(h', \varepsilon)\}$. Hence $\mathcal{S}(T) = [h]$ and $\mathcal{S}(T') = [h']$. Therefore, the result trivially holds.

Inductive step $N > 1$: Since a ChronTree is a binary tree, we can deduce that the number of leaves of a ChronTree is smaller than or equal to two power the height of the tree, hence $N \leq 2^{\mathsf{height}(T)}$ and $N' \leq 2^{\mathsf{height}(T')}$. Let's denote T_ℓ and T'_ℓ the respective subtrees of T and T' rooted by the left child of the root of T and T'. Similarly, let's denote T_r and T'_r the respective subtrees of T and T' rooted by the right child of the root of T and T'. By Definition B.3, we know that $\mathsf{height}(T_r) \leq \mathsf{height}(T_\ell)$, $\mathsf{height}(T'_r) \leq \mathsf{height}(T'_\ell)$ and both T_ℓ and T'_ℓ are perfect. Thus, we can deduce that $\mathsf{height}(T_\ell) = \mathsf{height}(T) - 1$, $\mathsf{height}(T'_\ell) = \mathsf{height}(T') - 1$, $\mathsf{height}(T_r) \leq \mathsf{height}(T) - 1$ and $\mathsf{height}(T'_r) \leq \mathsf{height}(T') - 1$. If we denote by $N_\ell, N'_\ell, N_r, N'_r$ the respective size of $T_\ell, T'_\ell, T_r, T'_r$, we deduce that:

— $N_\ell = 2^{\mathsf{height}(T)-1}$, $N'_\ell = 2^{\mathsf{height}(T')-1}$; and
— $N_r \leq 2^{\mathsf{height}(T)-1}$, $N'_r \leq 2^{\mathsf{height}(T')-1}$; and

— $N = N_\ell + N_r = N'_\ell + N'_r$.

Thus $2^{\mathsf{height}(T)-1} \leq N \leq 2^{\mathsf{height}(T)}$ and $2^{\mathsf{height}(T')-1} \leq N \leq 2^{\mathsf{height}(T')}$. Therefore, $\mathsf{height}(T) = \mathsf{height}(T')$. Since $\mathsf{height}(T) = \mathsf{height}(T')$, we deduce that $N_\ell = N'_\ell$ and $N_r = N'_r$. Hence we can apply our inductive hypothesis on T_ℓ, T'_ℓ and T_r, T'_r.

Assume that $h = h'$. Since T and T' are both ChronTree, we deduce that there exists $h_\ell, h'_\ell, h_r, h'_r$ respectively hash values of $T_\ell, T'_\ell, T_r, T'_r$ such that $h = \mathsf{h}(h_\ell, h_r)$ and $h' = \mathsf{h}(h'_\ell, h'_r)$. By the property of the hash function h, we deduce that $h_\ell, h_r = h'_\ell, h'_r$. But $h_\ell, h'_\ell, h_r, h'_r$ are all hash values, hence have the same size. Therefore $h_\ell, h_r = h'_\ell, h'_r$ implies $h_\ell = h'_\ell$ and $h_r = h'_r$. By our inductive hypothesis on both T_ℓ, T'_ℓ and T_r, T'_r, we deduce that for all $i \in \{1, \dots, N\}$, $d_i = d'_i$. The other side of the implication is trivial since h is determinist and by application of the inductive hypothesis on both T_ℓ, T'_ℓ and T_r, T'_r.

Let $i \in \{1, \dots, N\}$. Let $w \in \mathcal{A}^*$. Assume that $(d_i, w) \in T$. Note that $w \neq \varepsilon$ since $N > 1$. If $w = \ell \cdot w'$ (resp. $w = r \cdot w'$) for some w' then $(d_i, w') \in T_\ell$ (resp. $(d_i, w') \in T_r$). By our inductive hypothesis on T_ℓ and T'_ℓ (resp. T_r and T'_r), we deduce that $(d'_i, w') \in T'_\ell$ (resp. $(d'_i, w') \in T'_r$). Thus by definition of T'_ℓ (resp. T'_r), we conclude that $(d'_i, w) \in T'$. The other side of the equivalence is similar. □

To show that ChronTree can be an implementation of a chronological data structure, we need to define the different operations defined in Definition 3.1.

Definition B.4. Given a ChronTree T, we define $\mathsf{digest}_c(T)$, $\mathsf{content}_c(T)$, $\mathsf{add}_c(T,)$ as follows:

— $\mathsf{digest}_c(T) = (h, n)$ where h is the label of the root of T and $n = |\mathcal{S}(T)|$;
— $\mathsf{content}_c(T) = \mathcal{S}(T)$, that is the sequence of bitstrings stored in T;
— $\mathsf{add}_c(T, d) = \mathcal{CT}([d_1, \dots, d_k, d])$ where $\mathcal{S}(T) = [d_1, \dots, d_k]$.

Moreover, we define the $\mathsf{size}_c((h, n)) = n$ for some hash value h and integer n.

We now show the property of Definition 3.1 stating that two data structures with different contents imply different digests.

LEMMA B.3. *Let T and T' be two ChronTrees. We have* $\mathsf{content}_c(T) = \mathsf{content}_c(T')$ *if, and only if,* $\mathsf{digest}_c(T) \neq \mathsf{digest}_c(T')$.

PROOF. It is trivial from Definition B.4 and Lemma B.2. □

B.1. Positions in the ChronTree

We now define some functions linking the position of a leaf in a ChronTree and the number of data stored in the ChronTree.

Definition B.5. Let $\mathsf{Pos\_of\_Ind}$ be a function from $\mathbb{N}^+ \times \mathbb{N}^+$ to $\mathcal{A}^*$ defined recursively as follows:

— $\mathsf{Pos\_of\_Ind}(1, 1) = \varepsilon$
— $\mathsf{Pos\_of\_Ind}(i, N) = \ell \cdot \mathsf{Pos\_of\_Ind}(i, 2^{k-1})$ for some $k \in \mathbb{N}$ such that $i \leq 2^{k-1} < N \leq 2^k$
— $\mathsf{Pos\_of\_Ind}(i, N) = r \cdot \mathsf{Pos\_of\_Ind}(i - 2^{k-1}, N - 2^{k-1})$ for some $k \in \mathbb{N}$ such that $2^{k-1} < i \leq N \leq 2^k$.

Definition B.6. Let $\mathsf{Ind\_of\_Pos}$ be a function from $\mathcal{A}^* \times \mathbb{N}^+$ to $\mathbb{N}^+$ defined recursively as follows:

— $\mathsf{Ind\_of\_Pos}(\varepsilon, 1) = 1$
— $\mathsf{Ind\_of\_Pos}(\ell \cdot w, N) = \mathsf{Pos\_of\_Ind}(w, 2^{k-1})$ for some $k \in \mathbb{N}$ such that $2^{k-1} < N \leq 2^k$
— $\mathsf{Ind\_of\_Pos}(r \cdot w, N) = 2^{k-1} + \mathsf{Ind\_of\_Pos}(w, N - 2^{k-1})$ for some $k \in \mathbb{N}$ such that $2^{k-1} < N \leq 2^k$

Lastly, we define a function allowing us to prove that a position is a position of a node in a ChronTree.

Definition B.7 (*Verification of a position in ChronTree*). Let $N \in \mathbb{N}^+$ such that $2^{k-1} < N \leq 2^k$ for some $k \in \mathbb{N}$. Let $w \in \mathcal{A}^*$. The verification that w is a position of a node in a ChronTree of size N, denoted $\mathsf{VerifPos}_{CT}(w, N)$, is defined recursively as follows:

— $\mathsf{VerifPos}_{CT}(\varepsilon, N) = \mathsf{true}$;
— $\mathsf{VerifPos}_{CT}(w, N) = \mathsf{true}$ if $w = \ell \cdot w'$ for some $w' \in \mathcal{A}^*$ and $|w| \leq k$;
— $\mathsf{VerifPos}_{CT}(w, N) = \mathsf{VerifPos}_{CT}(w', N - 2^{k-1})$ if $w = r \cdot w'$ for some $w' \in \mathcal{A}^*$.

Basically, given a path w and an integer N, $\mathsf{VerifPos}_{CT}(w, N)$ verifies whether w is a valid path in a ChronTree T of size N.

Example B.2. Consider the T_c in Figure 6(c), if $w = \ell \cdot r \cdot r$ so $w = \ell \cdot w'$ such that $w' = r \cdot r$, then $\mathsf{VerifPos}_{CT}(w, N) = \mathsf{VerifPos}_{CT}(\ell \cdot r \cdot r, 6) = \mathsf{true}$ since $w' \in A^*$ and $|w| = 3$ and $2^2 < 6 \leq 2^3$. $\diamond$

LEMMA B.4. *Let T be a ChronTree with size N such that $\mathcal{S}(T) = [d_1, \dots, d_N]$. Let $i \in \{1, \dots, N\}$ and $w \in \mathcal{A}^*$. We have that :*

(*1*) *w is a position of a node in T iff $\mathsf{VerifPos}_{CT}(w, N) = \mathsf{true}$; and*
(*2*) *$\mathsf{Pos\_of\_Ind}(i, N)$ is the position of the leaf labeled by d_i in T; and*
(*3*) *if w is a position of a leaf in T then this leaf is labeled by d_i where $i = \mathsf{Ind\_of\_Pos}(w, N)$.*

PROOF. We prove the three properties by induction on N.

Base case $N = 1$: In such a case, T is a tree with only one node that is his root. Thus w is a position of a node in T is equivalent to $w = \varepsilon$ and so $\mathsf{VerifPos}_{CT}(w, N) = \mathsf{true}$. Moreover, $\mathsf{Pos\_of\_Ind}(1, 1) = \varepsilon$ by Definition B.5 with ε being the position of the only leaf. Lastly, $\mathsf{Ind\_of\_Pos}(\varepsilon, 1) = 1$ and ε is indeed the position of the leaf labeled by d_1. Hence the results hold.

Inductive step $N > 1$: Otherwise, let $k \in \mathbb{N}$ such that $2^{k-1} < N \leq 2^k$. We deduce that T is a tree with more than one node. Let us denote by T_ℓ and T_r the respective left and right child tree of the root of T. By definition of a ChronTree, we have that $\mathcal{S}(T_\ell) = [d_1, \dots, d_{k-1}]$ and $\mathcal{S}(T_r) = [d_{2^{k-1}+1}, \dots, d_N]$. Let us first compute $\mathsf{Pos\_of\_Ind}(i, N)$ and prove Property 2. We do a case analysis on i.

— Case $i \leq 2^{k-1}$: By Definition B.5, we have $\mathsf{Pos\_of\_Ind}(i, N) = \ell \cdot \mathsf{Pos\_of\_Ind}(i, 2^{k-1})$. But $\mathcal{S}(T_\ell) = [d_1, \dots, d_{2^{k-1}}]$ thus by inductive hypothesis, we deduce that $\mathsf{Pos\_of\_Ind}(i, 2^{k-1})$ is the position of the leaf, denoted n, labeled by d_i in T_ℓ. Moreover, T_ℓ is the left child tree of the root of T thus we deduce that the position of the leaf labeled by d_i in T is $\ell \cdot \mathsf{Pos\_of\_Ind}(i, 2^{k-1})$ that is $\mathsf{Pos\_of\_Ind}(i, N)$.
— Case $2^{k-1} < i \leq N$: By Definition B.5, we have $\mathsf{Pos\_of\_Ind}(i, N) = r \cdot \mathsf{Pos\_of\_Ind}(i - 2^{k-1}, N - 2^{k-1})$. But $\mathcal{S}(T_r) = [d_{2^{k-1}+1}, \dots, d_N]$ thus by inductive hypothesis on $N' = N - 2^{k-1}$, $i' = i - 2^{k-1}$ and for all $j \in \{1, \dots, N'\}$, $d'_j = d_{2^{k-1}+j}$, we deduce that $\mathsf{Pos\_of\_Ind}(i - 2^{k-1}, N - 2^{k-1})$ is the position of the leaf labeled by $d'_{i'}$ in T_r, that is the leaf labeled by $d_{2^{k-1}+i-2^{k-1}} = d_i$. At last, since T_r is the right child tree of the root of T, we can conclude that the position of the leaf labeled by d_i in T is $r \cdot \mathsf{Pos\_of\_Ind}(i - 2^{k-1}, N - 2^{k-1})$ that is $\mathsf{Pos\_of\_Ind}(i, N)$.

Let us now prove Property 3. We consider that w is a position of a leaf in T. Since $N > 1$, we know that $|w| > 1$. Thus, we do a case analysis on w.

— Case $w = \ell \cdot w'$ for some $w' \in \mathcal{A}^*$: Since T_ℓ is the left chid tree of the root of T, we deduce that w' is the position of a leaf in T_ℓ. But, by Definition B.7, $\mathsf{Ind\_of\_Pos}(w, N) = \mathsf{Pos\_of\_Ind}(w', 2^{k-1})$. Hence, with $\mathcal{S}(T_\ell) = [d_1, \dots, d_{2^{k-1}}]$, our inductive hypothesis gives us that w' is the position of the leaf labeled by d_i with $i = \mathsf{Pos\_of\_Ind}(w', 2^{k-1}) = \mathsf{Ind\_of\_Pos}(w, N)$.

— Case $w = r{\cdot}w'$ for some $w' \in \mathcal{A}^*$: Since T_r is the right child tree of the root of T, we deduce that w' is the position of the a leaf in T_r. But, by Definition B.7, $\mathsf{Ind\_of\_Pos}(w, N) = 2^{k-1} + \mathsf{Ind\_of\_Pos}(w', N - 2^{k-1})$. We know that $\mathcal{S}(T_r) = [d_{2^{k-1}+1}, \dots, d_N]$. By considering a renaming $d'_i = d_{2^{k-1}+i}$ for all $i \in \{1, \dots, N - 2^{k-1}\}$ and by applying our inductive hypothesis, we deduce that w' is the position of the leaf labeled by d'_i that is $d_{2^{k-1}+i}$ with $i = \mathsf{Ind\_of\_Pos}(w', N - 2^{k-1})$. Hence we deduce that w is the position of the leaf labeled by d_j with $j = 2^{k-1} + \mathsf{Ind\_of\_Pos}(w', N - 2^{k-1}) = \mathsf{Ind\_of\_Pos}(w, N)$. Hence the result holds.

Let us now focus on Property 1. We do a case analysis on w:

— Case $w = \varepsilon$: In such a case, w is the position of the root and by Definition B.7, $\mathsf{VerifPos}_{CT}(\varepsilon, N) = \mathsf{true}$ hence the result holds.

— Case $w = \ell \cdot w'$ for some $w' \in A^*$: w is the position of a node in T is equivalent to w' is a position of a node in T_ℓ. Moreover, by definition of a ChronTree of size N, we know that T_ℓ is perfect and of height $k - 1$. Hence w' is a position of a node in T_ℓ is equivalent to $|w'| \leq k - 1$. Therefore w is the position of a node in T is equivalent to $|w| \leq k$.

— Case $w = r \cdot w'$ for some $w' \in A^*$: w is the position of a node in T is equivalent to w' is a position of a node in T_r. Moreover, by definition of a ChronTree of size N, we know that T_r is of size $N - 2^{k-1}$. Hence by inductive hypothesis, $\mathsf{VerifPos}_{CT}(w', N - 2^{k-1}) = \mathsf{true}$ is equivalent to w' is a position of a node in T_r. Therefore, we can conclude that w is a position of a node in T iff $\mathsf{VerifPos}_{CT}(w, N) = \mathsf{true}$. □

B.2. Proof of presence

As previously mentioned, from a Merkle tree and so ChronTree, we can generate a proof that allows one to verify that some data is indeed in the tree.

Definition B.8 (*Proof of presence*). Given a bitstring d and a ChronTree T, the *proof of presence of d in T* exists if there is a node n_1 in T labeled by d; and is defined as $(w, [b_1, \dots, b_k])$ such that:

— w is the position of n_1 and $|w| = k$; and

— if $n_{k+1}, \dots, n_1$ is the path to n_1 in T then for all $i \in \{1, \dots, k\}$, b_i is the label of $\mathsf{Sib}_T(n_i)$.

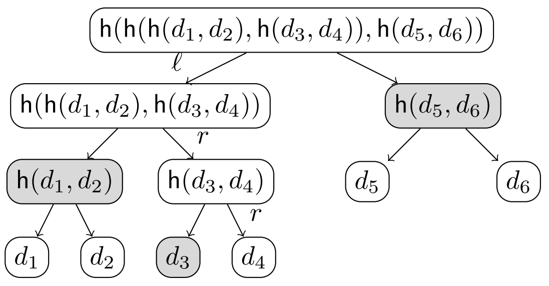
Typically, if (w, seq) is the proof of presence of some data d in a ChronTree T, then w represents the position of d in the ChronTree T and *seq* is the sequence of labels of the sibling of the nodes in the path of d. Intuitively, a proof of presence of d in T contains the minimum amount of information necessary to recompute the label of the root of T from the leaf containing d.

Example B.3. Consider the ChronTree T_c of Figure 6. The proof of presence of d_4 in T_c is the tuple (w, seq) where:

— $w = \ell \cdot r \cdot r$

— $\mathit{seq} = [d_3, \mathsf{h}(d_1, d_2), \mathsf{h}(d_5, d_6)]$

The proof of presence (w, seq) is graphically represented in Figure 7. The gray nodes are the nodes labeled by the elements of *seq* and the labeled vertices show the position w. ◇



$\mathsf{h}(\mathsf{h}(\mathsf{h}(d_1,d_2),\mathsf{h}(d_3,d_4)),\mathsf{h}(d_5,d_6))$
ℓ
$\mathsf{h}(\mathsf{h}(d_1,d_2),\mathsf{h}(d_3,d_4))$
$\mathsf{h}(d_5,d_6)$
r
$\mathsf{h}(d_1,d_2)$
$\mathsf{h}(d_3,d_4)$
d_5
d_6
r
d_1
d_2
d_3
d_4

Fig. 7. Proof of presence of d_4 in the ChronTree T_c

As we mentioned, a proof of presence is the minimum amount of information necessary to recompute the label of the root of T from the leaf containing d. Thus, we define the verification of a proof of presence for a ChronTree.

Definition B.9 (*Verification of a proof of presence*). Given data (w, seq) with $w \in \mathcal{A}^*$, and two bitstrings d and h, and N an integer. The verification that (w, seq) proves the presence of d in (h, N), denoted by $\mathsf{VerifPoP}_{CT}((h,N), d, (w, seq))$, is defined as follows: We have that $\mathsf{VerifPoP}_{CT}((h,N), d, (w, seq)) = \mathsf{true}$ if, and only if, $|w| = |seq|$, $\mathsf{VerifPos}_{CT}(w, N) = \mathsf{true}$, and $\mathsf{CompPoP}(w, seq, d) = h$, where $\mathsf{CompPoP}(w, seq, d)$ is defined recursively as follows.

— $\mathsf{CompPoP}(\varepsilon, [], t) = t$
— $\mathsf{CompPoP}(w \cdot \ell, [b_1, \ldots, b_k], t) = \mathsf{CompPoP}(w, [b_2, \ldots, b_k], \mathsf{h}(t, b_1))$
— $\mathsf{CompPoP}(w \cdot r, [b_1, \ldots, b_k], t) = \mathsf{CompPoP}(w, [b_2, \ldots, b_k], \mathsf{h}(b_1, t))$

Note that the size of such proof is at most proportional to the logarithm of the size of the ChronTree as stated in the following Lemma. Given a message m, we will denote by $\mathsf{size}(m)$ its size. Moreover, let's denote by $\mathsf{size}_{\mathsf{hash}}$ the size of the output of the hash function used in the ChronTree.

LEMMA B.5. *Let T be a ChronTree of size N and such that the size of the labels of the leaves is at most $\mathsf{size}_{\mathsf{data}}$. For all labels d of a node in T, if (w, seq) is the proof of presence of d in T then* $\mathsf{size}((w, seq)) \leq (1 + \max(\mathsf{size}_{\mathsf{data}}, \mathsf{size}_{\mathsf{hash}})) \times \lceil \ln(N) \rceil$.

PROOF. Since T is a ChronTree of size N, we have $2^{\mathsf{height}(T)-1} \leq N \leq 2^{\mathsf{height}(T)}$. Moreover, any path to a node in T is a sequence of at most $\mathsf{height}(T) + 1$ nodes, and the position w of this node is such that $|w| \leq \mathsf{height}(T)$.

Let n_1 be a node in T with $n_{k+1}, \ldots, n_1$ being the path to n_1 in T. According to Definition B.8, a proof of presence of n_1 in T is a tuple (w, seq) such that w is the position of n_1 and $seq = [b_1, \ldots, b_k]$ is such that for all $i \in \{1, \ldots, k\}$, b_i is the label of $\mathsf{Sib}_T(n_i)$. By definition of a tree, the sibling of a node can be either a leaf or a non-leaf node. Thus $\mathsf{size}(seq) \leq k \times \max(\mathsf{size}_{\mathsf{data}}, \mathsf{size}_{\mathsf{hash}})$. Moreover, T being a binary tree, w can be represented as a sequence of bit, that is $\mathsf{size}(w) \leq \mathsf{height}(T)$. Since $k \leq \mathsf{height}(T) \leq \lceil \ln(N) \rceil$, we conclude that $\mathsf{size}((w, seq)) \leq (1 + \max(\mathsf{size}_{\mathsf{data}}, \mathsf{size}_{\mathsf{hash}})) \times \lceil \ln(N) \rceil$. □

We also compute the the computation time of the verification of a proof of presence in a ChronTree. To do so, we will establish the complexity of our procedure with the size of the tree as parameter and considering that all basic operations such as addition, test of equality, etc as constant. However, since we also do compute several hash functions, we will also establish the number of computations of a hash during the verification.

LEMMA B.6. *Let T be a ChronTree of size N and such that the size of the labels of the leaves is at mode $\mathsf{size}_{\mathsf{data}}$. For all labels d of a node in T, if (w, seq) is the proof of* *presence of d in T and the computation time of* $\mathsf{VerifPoP}_{CT}((h,N),d,(w,seq))$ *is* $O(\ln(N))$ *and requires at most* $\lceil \ln(N) \rceil$ *computations of a hash on some data of size at most* $2 \times \max(\mathsf{size}_{\mathsf{data}}, \mathsf{size}_{\mathsf{hash}})$*.*

PROOF. The computation of $\mathsf{VerifPoP}_{CT}((h,N),d,(w,seq))$ consists of the computation of $\mathsf{VerifPos}_{CT}(w,N)$ and $\mathsf{CompPoP}(w,seq,d)$. But $\mathsf{VerifPos}_{CT}(w,N)$ is recursively defined such that the recursive step is given by $\mathsf{VerifPos}_{CT}(w,N) = \mathsf{VerifPos}_{CT}(w',N-2^{k-1})$ if $w = r \cdot w'$ for some $w' \in \mathcal{A}^*$. Hence we trivially deduce that the computation time of $\mathsf{VerifPos}_{CT}(w,N)$ is $O(|w|)$ and we already proved that $|w| \leq \lceil \ln(N) \rceil$. Hence the computation time of $\mathsf{VerifPos}_{CT}(w,N)$ is $O(\ln(N))$. Similarly $\mathsf{CompPoP}(w,seq,t)$ is recursively defined as follows:

— $\mathsf{CompPoP}(\varepsilon, [], t) = t$
— $\mathsf{CompPoP}(w \cdot \ell, [b_1,\dots,b_k], t) = \mathsf{CompPoP}(w, [b_2,\dots,b_k], \mathsf{h}(t,b_1))$
— $\mathsf{CompPoP}(w \cdot r, [b_1,\dots,b_k], t) = \mathsf{CompPoP}(w, [b_2,\dots,b_k], \mathsf{h}(b_1,t))$

where $seq = [b_1,\dots,b_k]$. Hence we deduce that their is at most $|w|$ recursive call and at each call, we compute a hash function on two terms that are either a hash value or the label of a leaf. Therefore, the computation of $\mathsf{CompPoP}(w,seq,d)$ is $O(\ln(N))$ and requires at most $\lceil \ln(N) \rceil$ computations of a hash on some data of size at most $2 \times \max(\mathsf{size}_{\mathsf{data}}, \mathsf{size}_{\mathsf{hash}})$. This allows us to conclude that the computation time of $\mathsf{VerifPoP}_{CT}((h,N),d,(w,seq))$ is $O(\ln(N))$ and requires at most $\lceil \ln(N) \rceil$ computations of a hash on some data of size at most $2 \times \max(\mathsf{size}_{\mathsf{data}}, \mathsf{size}_{\mathsf{hash}})$. □

LEMMA B.7. *Let T be a ChronTree of size N with h as hash value. Let $w \in \mathcal{A}^*$, seq be a sequence of bitstrings and let d be a bitstring. We have that* $\mathsf{VerifPoP}_{CT}((h,N),d,(w,seq)) = \mathsf{true}$ *if, and only if, $(d,w) \in T$ and (w,seq) is a proof of presence of d in T.*

PROOF. We prove this lemma by induction on $|w|$.

Base case $|w| = 0$: In such a case, we have $w = \varepsilon$. We prove the two sides of the equivalence.
Assume first that $\mathsf{VerifPoP}_{CT}((h,N),d,(w,seq)) = \mathsf{true}$. By definition B.9, we deduce that $|w| = |seq| = 0$, $\mathsf{VerifPos}_{CT}(w,N) = \mathsf{true}$, and $\mathsf{CompPoP}(w,seq,d) = h$. But $|seq| = 0$ implies that $seq = [\ ]$ and so $\mathsf{CompPoP}(\varepsilon,[],d) = d$. Therefore, $d = h$ and since $(h,\varepsilon) \in T$, we deduce that $(d,w) \in T$.
Let's now compute the proof of presence of d in T. We just have shown that $(d,w) \in T$. thus according to Definition B.8, the proof of presence is $(w,[b_1,\dots,b_k])$ where $k = |w|$. Since $|w| = 0$, we deduce that $[b_1,\dots,b_k] = [\ ]$ and so (w,seq) is the proof of presence of d in T.

Assume now that $(d,w) \in T$ and (w,seq) is the proof of presence of d in T. By Definition B.8, $|w| = |seq|$ thus $seq = [\ ]$. We want to show that $\mathsf{VerifPoP}_{CT}((h,N),d,(w,seq)) = \mathsf{true}$, thus it remains to prove that $\mathsf{VerifPos}_{CT}(w,N) = \mathsf{true}$ and $\mathsf{CompPoP}(w,seq,d) = h$. But $(d,w) \in T$ implies that w is a position in the tree T and by Lemma B.4, $\mathsf{VerifPos}_{CT}(w,N) = \mathsf{true}$. At last, by Definition B.9, we have $\mathsf{CompPoP}(\varepsilon,[],d) = d$. But we $(d,\varepsilon) \in T$ and $(h,\varepsilon) \in T$ by hypothesis. Thus since T is a binary tree, we deduce that $d = h$ and $\mathsf{CompPoP}(w,seq,d) = h$. Hence the result holds.

Inductive step $|w| > 0$: Otherwise we have that $w = w' \cdot a$ with $w' \in \mathcal{A}^*$ and $a \in \mathcal{A}$. Once again, we prove the two sides of the equivalence.
Assume first that $\mathsf{VerifPoP}_{CT}((h,N),d,(w,seq)) = \mathsf{true}$. By Definition B.9, we deduce that $|w| = |seq|$, $\mathsf{VerifPos}_{CT}(w,N) = \mathsf{true}$, and $\mathsf{CompPoP}(w,seq,d) = h$. Let's denote $seq = [b_1,\dots,b_k]$. By Lemma B.4, we know that $\mathsf{VerifPos}_{CT}(w,N) = \mathsf{true}$ implies that w is a position in the tree T and so w' is also a position in T. Once again by Lemma B.4, we deduce that $\mathsf{VerifPos}_{CT}(w',N) = \mathsf{true}$. Moreover, if we denote $seq' = [b_2,\dots,b_k]$,

we have that $|seq'| = |w'|$. At last, thanks to Definition B.9, $\mathsf{CompPoP}(w, seq, d) = \mathsf{CompPoP}(w', seq', \mathsf{h}(d, b_1))$ if $a = \ell$, $\mathsf{CompPoP}(w, seq, d) = \mathsf{CompPoP}(w', seq', \mathsf{h}(b_1, t))$ otherwise. Thus, we deduce that if $a = \ell$ (resp. $a = r$) then (w', seq') proves the presence of $\mathsf{h}(d, b_1)$ (resp. $\mathsf{h}(b_1, d)$) in (h, N), that is $\mathsf{VerifPoP}_{CT}((h, N), \mathsf{h}(d, b_1), (w', seq')) = \mathsf{true}$ (resp. $\mathsf{VerifPoP}_{CT}((h, N), \mathsf{h}(b_1, d), (w', seq')) = \mathsf{true}$). By inductive hypothesis on w', we can deduce that $(\mathsf{h}(d, b_1), w') \in T$ (resp. $(\mathsf{h}(b_1, d), w') \in T$). But we already proved that w is a valid position in T, thus since T is a ChronTree and $w = w' \cdot \ell$ (resp. $w = w' \cdot r$), we can deduce that $(d, w) \in T$.

Let's now compute the proof of presence of d in T. We just have shown that $(d, w) \in T$ and $(\mathsf{h}(d, b_1), w') \in T$ (resp. $(\mathsf{h}(b_1, d), w') \in T$). Thus our inductive hypothesis, we deduce that (w', seq') is the proof of $\mathsf{h}(d, b_1)$ (resp. $\mathsf{h}(b_1, d)$) in T. Thus, we have that if $n_2, \ldots n_{k+1}$ is the path of $n_2 = (\mathsf{h}(d, b_1), w')$ (resp. $n_2 = (\mathsf{h}(b_1, d), w')$) in T then for all $i \in \{2, \ldots, k\}$, b_i is the label of the sibling of n_i in T. But we know that $(d, w) \in T$ thus by definition of a ChronTree, $(\mathsf{h}(d, b_1), w') \in T$ (resp. $(\mathsf{h}(b_1, d), w') \in T$) implies that b_1 is the label of the sibling of (w, d). Hence (w, seq) is the proof of presence of d in T, and so the result holds.

Assume now that $(d, w) \in T$ and (w, seq) is the proof of presence of d in T. By Lemma B.4, we directly have that $\mathsf{VerifPos}_{CT}(w, N) = \mathsf{true}$. Moreover, by Definition B.8, we deduce that $|w| = |seq|$. Hence it remains to compute $\mathsf{CompPoP}(w, seq, d)$. Let's denote $seq = [b_1, \ldots, b_k]$. If $a = \ell$ (resp. $a = r$) then by Definition B.8, we have that $(w', \mathsf{h}(d, b_1)) \in T$ (resp. $(w', \mathsf{h}(b_1, d)) \in T$). Moreover, we also deduce that $(w', [b_2, \ldots, b_k])$ is the proof of presence of $\mathsf{h}(d, b_1)$ (resp. $\mathsf{h}(b_1, d)$) in T. Thus by our inductive hypothesis, we have that $(w', [b_2, \ldots, b_k])$ proves the presence of $\mathsf{h}(d, b_1)$ (resp. $\mathsf{h}(b_1, d)$) in (h, N). Hence $\mathsf{CompPoP}(w', seq', \mathsf{h}(d, b_1)) = h$ (resp. $\mathsf{CompPoP}(w', seq', \mathsf{h}(b_1, d)) = h$) where $seq' = [b_2, \ldots, b_k]$. Hence by Definition B.9, we conclude that $\mathsf{CompPoP}(w, seq, d) = h$ and so the result holds. $\square$

With these lemmas, we can define the procedure $\mathsf{VerifPoP}_c$ for our implementation of chronological data structure with ChronTrees, and show that it satisfies the properties described in Definition 3.1. With the boolean values true and false, we consider the classical logic operators $\wedge, \vee$, the (dis)equality tests $=^?, \neq^?$ and the inequality tests $\leq^?, \geq^?$.

Definition B.10. Let X be a set of data. Let T be a ChronTree and let $(h, N) = \mathsf{digest}_c(T)$. Let $d \in X$ and $n \in \mathbb{N}$. Let $w \in \mathcal{A}^*$ and seq be a sequence of bitstring. We define $\mathsf{VerifPoP}_c((h, N), d, n, (w, seq)) = \mathsf{VerifPoP}_{CT}((h, N), d, (w, seq)) \wedge \mathsf{Pos\_of\_Ind}(n, N) =^? w \wedge 1 \leq^? n \leq^? N$.

LEMMA B.8. *Let X be a set of data. Let T be a ChronTree and let $(h, N) = \mathsf{digest}_c(T)$. For all $d \in X$, for all $n \in \mathbb{N}$, we have that d is the n^{th} element of $\mathsf{content}_c(T)$ if, and only if, there exists (w, seq) of size $O(\log(|\mathsf{content}_c(T)|))$ such that $\mathsf{VerifPoP}_c(dg, d, n, (w, seq)) =$ true. Moreover, the computation time of $\mathsf{VerifPoP}_c$ is linear in the size of its inputs.*

PROOF. By definition, we have that $\mathsf{content}_c(T) = \mathcal{S}(T)$. By Definition B.4, if $(h, N) = \mathsf{digest}_c(T)$ then h is the label of the root of T and N is the size of T, that is the number of leaves in T. Let's denote $\mathsf{content}_c(T) = [d_1, \ldots, d_N]$. Moreover, let $d \in X$ and $n \in \mathbb{N}$.

Assume that d is the n^{th} element of $\mathsf{content}_c(T) = \mathcal{S}(T)$, that is equivalent to $d = d_n$ is the label of a leaf in T and $1 \leq n \leq N$. By Definition B.8, we deduce that the proof of presence of d_n in T exists and denoted (w, seq) where w is the position of the node labeled by d_n in T. By Lemma B.7, we deduce $\mathsf{VerifPoP}_{CT}((h, N), d_n, (w, seq)) = \mathsf{true}$. At last by Lemma B.4, we know that $\mathsf{Pos\_of\_Ind}(n, N)$ is the position of the leaf labeled by d_n in T since $\mathcal{S}(t) = [d_1, \ldots, d_N]$. Hence it implies that $\mathsf{Pos\_of\_Ind}(n, N) = w$. Moreover, by Lemma B.5, we know (w, seq) is of size $O(\log(N))$. At last, by Lemma B.6, we know that the computation time of $\mathsf{VerifPoP}_{CT}((h, N), d, (w, seq))$ is $O(\ln(N))$. Since (w, seq) is of size $O(\log(N))$ and h, d are $O(1)$, we deduce that the computation time of $\mathsf{VerifPoP}_c$ is linear in the size of its inputs.

Let us assume now that there exists (w, seq) such that $\mathsf{VerifPoP}_c(dg, d, n, (w, \mathit{seq})) = \mathsf{true}$, that is $\mathsf{VerifPoP}_{CT}((h, N), d, (w, \mathit{seq})) = \mathsf{true}$, $\mathsf{Pos\_of\_Ind}(n, N) = w$ and $1 \leq n \leq N$. By Lemma B.7, we deduce that $(w, d) \in T$. By Lemma B.4, we know that $\mathsf{Pos\_of\_Ind}(n, N)$ is the position of the leaf labeled by d_n in T since $\mathcal{S}(t) = [d_1, \ldots, d_N]$. Hence, with $\mathsf{Pos\_of\_Ind}(n, N) = w$, we conclude that d is the n^{th} element of $\mathsf{content}_c(T)$. □

B.3. Proof of extension

We show in this section how we implement the procedure $\mathsf{VerifPoE}_c$ allowing us to prove the extension between two digests of chronological data structures. In the binary representation of a number, we consider that the rightmost bit is at position 0. For example in 01001100, the smallest position of the bits 1 is 2.

Definition B.11 (*Proof of extension*). Let X be a set. Let T' and T be ChronTrees of size $N' < N$ respectively, containing the data $\mathcal{S}(T) = [d_1, \ldots, d_{N'}, \ldots, d_N]$ and $\mathcal{S}(T') = [d_1, \ldots, d_{N'}]$ for $d_1, \ldots, d_{N'}, \ldots, d_N \in X$. Let m be the smallest position of the bits 1 in the binary representation of N'. Let $w, w' \in \mathcal{A}^*$ such that $\mathsf{Pos\_of\_Ind}(N', N) = w \cdot w'$ with $|w'| = m$. Let d such that $(w, d) \in T$. At last, let (w, seq') be the proof of presence of d in T. The proof of extension of T' into T is defined as the sequence seq of bitstrings such that

— if $N' = 2^k$ for some k, then $\mathit{seq} = \mathit{seq}'$; otherwise
— $\mathit{seq} = d :: \mathit{seq}'$, where :: is the concatenation operation.

While a proof of presence is the minimal amount of information needed to recompute the root value of a ChronTree from the label of a node, the proof of extension is the minimal amount of information needed to recompute the hash value of ChronTree T from the hash value of a ChronTree T' where T is an extension of T'. Intuitively, if T' of size N' is a perfect tree i.e. $N' = 2^k$ for some k, then the root of T' is a node in T. So the proof of extension of a ChronTree T' into a ChronTree T is the proof that the root value of T' is in T. If T' is not perfect, then the proof of extension of a ChronTree T' into a ChronTree T is the proof that the root value of the largest perfect subtree containing $d_{N'}$ in T' is also in T, where $\mathcal{S}(T') = [d_1, \ldots, d_{N'}]$. With such a proof and the size of both trees, we can reconstruct the root value of T and T' as the verification of the proof that T is extended from T'.

Example B.4. Coming back to the ChronTrees in Figure 6, the proof of extensions of:

— T_a into T_b is the sequence $\mathit{seq}_{ab} = [d_3, d_4, \mathsf{h}(d_1, d_2)]$;
— T_a into T_c is the sequence $\mathit{seq}_{ac} = [d_3, d_4, \mathsf{h}(d_1, d_2), \mathsf{h}(d_5, d_6)]$;
— T_b into T_c is the sequence $\mathit{seq}_{bc} = [\mathsf{h}(d_5, d_6)]$. ◇

As previously mentioned, the size of the corresponding ChronTrees are needed for the verification of the proof of extension.

Definition B.12 (*Verification of a proof of extension*). Let $N, N' \in \mathbb{N}^*$. Let h, h' be two bitstrings and let $\mathit{seq} = [b_1, \ldots, b_k]$ be a sequence of bitstrings. Let m be the smallest position of the bits 1 in the binary representation of N'. The verification of seq proving the extension of h' of size N' into h of size N, denoted by $\mathsf{VerifPoE}_{CT}(h, N, h', N', \mathit{seq})$, is defined as follows:

We have that $\mathsf{VerifPoE}_{CT}(h, N, h', N', \mathit{seq}) = \mathsf{true}$ if, and only if, $N' < N$ and if $\mathsf{Pos\_of\_Ind}(N', N) = w \cdot w'$ for some w, w' with $|w'| = m$ then

— if $N' = 2^{k'}$ for some $k' \in \mathbb{N}$ then $\mathsf{VerifPoP}_{CT}((h, N), h', (w, \mathit{seq})) = \mathsf{true}$;

— otherwise $|w| = k-1$ and if we denote $w = a_k \cdot \ldots \cdot a_2$ and if $S = [i_1, \ldots, i_p]$ is the increasing sequence of integer such that $\forall i \in S$, $a_i = r$; and $\forall i \in \{2, \ldots, k\} \smallsetminus S$, $a_i = \ell$ then we have that $\mathsf{VerifPoP}_{CT}((h, N), b_1, (w, [b_2, \ldots, b_k])) = \mathsf{true}$ and $\mathsf{VerifPoP}_{CT}((h', N'), b_1, (w_r, [b_{i_1}, \ldots, b_{i_p}])) = \mathsf{true}$ where $\mathsf{Pos\_of\_Ind}(N', N') = w_r \cdot w'$.

Example B.5. Coming back to Example B.4, consider $h_a = \mathsf{h}(\mathsf{h}(d_1, d_2), d_3)$, $h_b = \mathsf{h}(\mathsf{h}(d_1, d_2), \mathsf{h}(d_3, d_4))$ and $h_c = \mathsf{h}(\mathsf{h}(\mathsf{h}(d_1, d_2), \mathsf{h}(d_3, d_4)), \mathsf{h}(d_5, d_6))$. We have that:

— seq_{ab} proves the extension of h_a of size 3 into h_b of size 4;
— seq_{ac} proves the extension of h_a of size 3 into h_c of size 6;
— seq_{bc} proves the extension of h_b of size 4 into h_c of size 6;

Figure 8 is the graphical representation of the verification of seq_{ac} given h_a and h_c. In particular, $(\ell \cdot r \cdot \ell, [d_4, \mathsf{h}(d_1, d_2), \mathsf{h}(d_5, d_6)])$ proves the presence of d_3 in h_c and $(r, [\mathsf{h}(d_1, d_2)])$ proves the presence of d_3 in h_a. $\diamond$

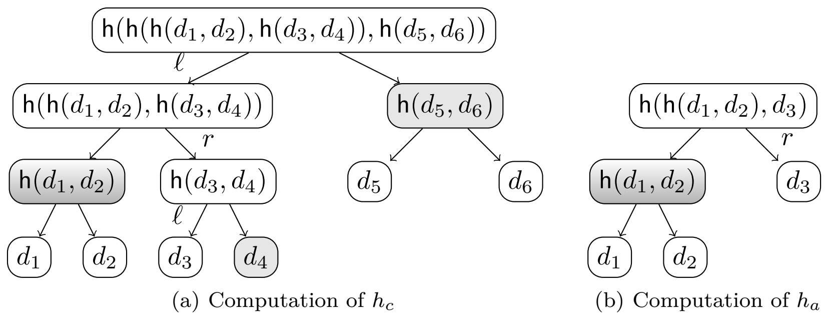


Fig. 8. Verification of the proof of extension seq_{ac} given h_a and h_c

Similarly to the proof of presence, the size of a proof of extension is logarithmic to the size of the extended ChronTree. Note that this result is expected since a proof of extension is in fact a proof of presence of some data in the extended tree.

LEMMA B.9. *Let T be a ChronTree of size N and T' a ChronTree of size N' such that T is an extension of T' and the size of the labels of the leaves is at most $\mathsf{size}_{\mathsf{data}}$. If seq is the proof of extension of T' into T then $\mathsf{size}(seq) \leq \max(\mathsf{size}_{\mathsf{data}}, \mathsf{size}_{\mathsf{hash}}) \times (\lceil \ln(N) \rceil + 1)$.*

PROOF. From Definition B.11, we have that (w, seq') is a proof of presence of some d in T, and either $seq = seq'$ or $seq = d :: seq'$. By Lemma B.5, we know that $\mathsf{size}((w, seq')) \leq 1 + \max(\mathsf{size}_{\mathsf{data}}, \mathsf{size}_{\mathsf{hash}})) \times \lceil \ln(N) \rceil$ and in particular $seq' \leq \max(\mathsf{size}_{\mathsf{data}}, \mathsf{size}_{\mathsf{hash}})) \times \lceil \ln(N) \rceil$. Moreover, since $\mathsf{size}(d) \leq \max(\mathsf{size}_{\mathsf{data}}, \mathsf{size}_{\mathsf{hash}}))$, we deduce that $\mathsf{size}(seq) \leq \max(\mathsf{size}_{\mathsf{data}}, \mathsf{size}_{\mathsf{hash}}) \times (\lceil \ln(N) \rceil + 1)$. $\square$

We also compute the the computation time of the verification of a proof of extension in a ChronTree. As for the verification of a proof of presence, we also establish the number of computations of a hash during the verification.

LEMMA B.10. *Let T be a ChronTree of size N and T' a ChronTree of size N' such that T is an extension of T' and the size of the labels of the leaves is at most $\mathsf{size}_{\mathsf{data}}$. For all sequence seq, if seq is the proof of extension of T' into T and the computation time of $\mathsf{VerifPoE}_{CT}(h, N, h', N', seq)$ is $O(\ln(N))$ and requires at most $2 \times \lceil \ln(N) \rceil$ computations of a hash on some data of size at most $2 \times \max(\mathsf{size}_{\mathsf{data}}, \mathsf{size}_{\mathsf{hash}})$.*

PROOF. Assume that $seq = [b_1, \ldots, b_k]$. According to Definition B.12, the computation of $\mathsf{VerifPoE}_{CT}(h, N, h', N', seq)$ consists of the computation of $\mathsf{Pos\_of\_Ind}(N', N)$ and either the computation of $\mathsf{VerifPoP}_{CT}((h, N), h', (w, seq))$ or the computations of $\mathsf{VerifPoP}_{CT}((h', N'), b_1, (w_r, [b_{i_1}, \ldots, b_{i_p}]))$, $\mathsf{Pos\_of\_Ind}(N', N)$ and $\mathsf{VerifPoP}_{CT}((h, N), b_1, (w, [b_2, \ldots, b_k]))$.
Since $N' < N$ and by Lemma B.6, we deduce that the computation time of $\mathsf{VerifPoP}_{CT}((h, N), h', (w, seq))$ (resp. $\mathsf{VerifPoP}_{CT}((h, N), b_1, (w, [b_2, \ldots, b_k]))$ and $\mathsf{VerifPoP}_{CT}((h', N'), b_1, (w_r, [b_{i_1}, \ldots, b_{i_p}])))$ is $O(\ln(N))$ and requires at most $\lceil \ln(N) \rceil$ computations of a hash on some data of size at most $2 \times \max(\mathsf{size_{data}}, \mathsf{size_{hash}})$. Moreover, the computation time of $\mathsf{Pos\_of\_Ind}(N', N)$ and $\mathsf{Pos\_of\_Ind}(N', N)$ are $O(\ln(N))$. Thus, we can conclude that the computation time of $\mathsf{Pos\_of\_Ind}(N', N)$ is $O(\ln(N))$ and requires at most $2 \times \lceil \ln(N) \rceil$ computations of a hash on some data of size at most $2 \times \max(\mathsf{size_{data}}, \mathsf{size_{hash}})$. □

It remains for us to prove the soundness of the procedure $\mathsf{VerifPoP}_{CT}$. But to do so, we first need to prove some preliminaries result on the size of the path in a ChronTree.

LEMMA B.11. *Let $N \in \mathbb{N}^*$ and let $N' \in \{1, \ldots, N\}$. Let m be the smallest position of bits 1 in the binary representation of N'. There exists $w, w' \in \mathcal{A}^*$ such that $\mathsf{Pos\_of\_Ind}(N', N) = w \cdot w'$, $w' = r \cdot \ldots \cdot r$ and $|w'| = m$. Moreover, $w = \varepsilon$ if and only if $N' = N = 2^k$. At last, if $N' = 2^k$ for some $k \in \mathbb{N}$ then $w = \ell \cdot \ldots \cdot \ell$ (w can be ε).*

PROOF. We prove the result by induction on N.

Base case $N = 1$: In such a case, $N' = 1$ and so $m = 0$. Moreover, $\mathsf{Pos\_of\_Ind}(N', N) = \varepsilon$ and so the result trivially holds.

Inductive step $N > 1$: Assume first that $N' \leq 2^{k-1} < N \leq 2^k$. In such a case, we have that $\mathsf{Pos\_of\_Ind}(N', N) = \ell \cdot \mathsf{Pos\_of\_Ind}(N', 2^{k-1})$. Hence by inductive hypothesis, we deduce that $\mathsf{Pos\_of\_Ind}(N', 2^{k-1}) = w \cdot w'$, $w' = r \cdot \ldots \cdot r$, $|w'| = m$ and if $N' = 2^q$ for some $q \in \mathbb{N}$ then $w = \ell \cdot \ldots \cdot \ell$. Hence $\mathsf{Pos\_of\_Ind}(N', N) = (\ell \cdot w) \cdot w'$, $w' = r \cdot \ldots \cdot r$, $|w'| = m$ and if $N' = 2^q$ for some $q \in \mathbb{N}$ then $\ell \cdot w = \ell \cdot \ldots \cdot \ell$; and so the result holds. Note that $(\ell \cdot w) \neq \varepsilon$.
Otherwise we have that $2^{k-1} < N' \leq N \leq 2^k$ and so $\mathsf{Pos\_of\_Ind}(N', N) = r \cdot \mathsf{Pos\_of\_Ind}(N' - 2^{k-1}, N - 2^{k-1})$. But $2^{k-1} < N' \leq N \leq 2^k$ implies that either (a) $2^{k-1} < N' < 2^k$ or else (b) $N' = N = 2^k$. In case (a), $m < k - 1$ and so the smallest position of bits 1 in the binary representation of $N' - 2^{k-1}$ is m. Thus by inductive hypothesis, $\mathsf{Pos\_of\_Ind}(N' - 2^{k-1}, N - 2^{k-1}) = w \cdot w'$, $w' = r \cdot \ldots \cdot r$ and $|w'| = m$. This implies that $\mathsf{Pos\_of\_Ind}(N', N) = (r \cdot w) \cdot w'$, $w' = r \cdot \ldots \cdot r$ and $|w'| = m$. Note that $N' \neq 2^q$ for all $q \in \mathbb{N}$ and $(r \cdot w) \neq \varepsilon$.
In case (b), the smallest position of bits 1 in the binary representation of $N' - 2^{k-1} = 2^{k-1}$ is $k - 1$. Thus by inductive hypothesis, $\mathsf{Pos\_of\_Ind}(N' - 2^{k-1}, N - 2^{k-1}) = w'$, $w' = r \cdot \ldots \cdot r$ and $|w'| = k - 1$. We can deduce that $\mathsf{Pos\_of\_Ind}(N', N) = r \cdot w' = r \cdot \ldots \cdot r$ with $|r \cdot w'| = k$. Hence the result holds. □

LEMMA B.12. *Let T be s ChronTree of size N. Let's denote $\mathcal{S}(T) = [d_1, \ldots, d_N]$. Let $N = \sum_{i=1}^{n} 2^{a_i}$ where $a_1, \ldots, a_n$ is a strictly decreasing sequence of integer. Let $N' \in \{1, \ldots, N\}$, the path of the leaf labeled by $d_{N'}$ is of size $m \in \mathbb{N}$ where:*

— *if $1 + \sum_{i=1}^{k-1} 2^{a_i} \leq N' \leq \sum_{i=1}^{k} 2^{a_i}$ for some $k \in \{1, \ldots, n-1\}$ then $m = a_k + k$*
— *otherwise $1 + \sum_{i=1}^{n-1} 2^{a_i} \leq N' \leq \sum_{i=1}^{n} 2^{a_i}$ and $m = a_n + n - 1$*

PROOF. We do a proof by induction on the height of T.

Base case $\mathsf{height}(T) = 0$: In such a case, $T = \{(d_1, \varepsilon)\}$, $N = 1$ and so $N' = 1 = 2^0$. We trivially have that $m = 0 = 0 + 1 - 1$ hence the result holds.

Inductive step $\mathsf{height}(T) > 0$*:* Let $N = \sum_{i=1}^{n} 2^{a_i}$. Let $N' \in \{1, \ldots, N\}$. By definition of a ChronTree, we know that the subtree T_ℓ rooted by the left child of the root of T is perfect and its height is $\mathsf{height}(T)-1$. Moreover, with T being full, we deduce that $N > 2^{\mathsf{height}(T)-1}$.

Let's first assume that T is perfect. In such a case, we have that $N = 2^{\mathsf{height}(T)}$ and $n = 1$ with $a_1 = \mathsf{height}(T)$. Thus we do have that $m = a_1 + 1 - 1 = a_1$.

Else if T is not perfect then we can deduce that $2^{\mathsf{height}(T)-1} < N < 2^{\mathsf{height}(T)}$. Thus we can deduce that $a_1 = \mathsf{height}(T) - 1$. Moreover since T_ℓ is perfect, we can deduce that if $1 \leq N' \leq 2^{a_1}$ then $m = \mathsf{height}(T) = a_1 + 1$.

We also know that the subtree T_r rooted by the right child of the root of T is a ChronTree such that $\mathsf{height}(T_r) \leq \mathsf{height}(T)-1$, $\mathcal{S}(T_r) = [d_{2^{a_1}+1}, \ldots, d_N]$ and is of size $\sum_{i=2}^{n} 2^{a_i}$. Thus by applying our inductive hypothesis on T_r, we deduce that for all $N' \in \{2^{a_1}+1; \ldots; N\}$, the path of the leaf labeled by $d_{N'}$ in T_r is of size $m' \in \mathbb{N}$ where:

— if $1+\sum_{i=1}^{k-1} 2^{a_{i+1}} \leq N'-2^{a_1} \leq \sum_{i=1}^{k} 2^{a_{i+1}}$ for some $k \in \{1, \ldots, n-2\}$ then $m' = a_{k+1}+k$
— otherwise $1 + \sum_{i=1}^{n-2} 2^{a_{i+1}} \leq N' - 2^{a_1} \leq \sum_{i=1}^{n-1} 2^{a_{i+1}}$ and $m' = a_n + n - 2$

Since the size of path of a leaf in T_r is the size of the path of this same leaf in T minus one, we can deduce that for all $N' \in \{2^{a_1}+1; \ldots; N\}$, the path of the leaf labeled by $d_{N'}$ in T is of size $m \in \mathbb{N}$ where:

— if $1 + \sum_{i=1}^{k} 2^{a_i} \leq N' \leq \sum_{i=1}^{k+1} 2^{a_i}$ for some $k \in \{1, \ldots, n-2\}$ then $m = a_{k+1} + k + 1$
— otherwise $1 + \sum_{i=1}^{n-1} 2^{a_i} \leq N' \leq \sum_{i=1}^{n} 2^{a_i}$ and $m = a_n + n - 1$

Since we already proved that if $1 \leq N' \leq 2^{a_1}$ then $m = \mathsf{height}(T) = a_1+1$, we can conclude that for all $N' \in \{1; \ldots; N\}$, the path of the leaf labeled by $d_{N'}$ in T is of size $m \in \mathbb{N}$ where:

— if $1 + \sum_{i=1}^{k-1} 2^{a_i} \leq N' \leq \sum_{i=1}^{k} 2^{a_i}$ for some $k \in \{1, \ldots, n-1\}$ then $m = a_k + k$
— otherwise $1 + \sum_{i=1}^{n-1} 2^{a_i} \leq N' \leq \sum_{i=1}^{n} 2^{a_i}$ and $m = a_n + n - 1$

Hence the result holds. □

LEMMA B.13. *Let $N \in \mathbb{N}$ and let $N' \leq N$. Let m be the smallest position of bits* 1 *in the binary representation of N'. We have that $|\mathsf{Pos\_of\_Ind}(N', N)| \geq m$.*

PROOF. Let $a_1, \ldots, a_n$ be a strictly decreasing sequence of integers such that $N = \sum_{i=1}^{n} 2^{a_i}$. Since $N' < N$, we know that there exists $k \in \{1, \ldots, n\}$ such that $1+\sum_{i=1}^{k-1} 2^{a_i} \leq N' \leq \sum_{i=1}^{k} 2^{a_i}$. That is $1+\sum_{i=1}^{k-1} 2^{a_i} \leq N' \leq 2^{a_k}+\sum_{i=1}^{k-1} 2^{a_i}$. Since $a_1, \ldots, a_n$ are strictly decreasing, we deduce that $m \leq a_k$. But by Lemma B.12, we know that $|\mathsf{Pos\_of\_Ind}(N', N)| \geq a_k + k - 1$, meaning that $|\mathsf{Pos\_of\_Ind}(N', N)| \geq m$. □

LEMMA B.14. *Let T be a ChronTree of size N with h as hash value. Let $N' \in \mathbb{N}^*$ such that $N' < N$. Let seq be a sequence of bitstrings and h' be a bitstring. We have that* $\mathsf{VerifPoE}_{CT}(h, N, h', N', seq) = \mathsf{true}$ *if, and only if, there exists a ChronTree T' of size N' with h' as hash value such that T is an extension of T' and seq is the proof of extension of T' into T.*

PROOF. We prove the result by induction on N.

Base case $N = 1$*:* This case trivially holds since N' cannot be in $\mathbb{N}^*$ and $N' < N$.

Inductive case $N > 1$*:* In this case, we have that $\mathcal{S}(T) = [d_1, \ldots, d_N]$. Let m be the smallest position of the bits 1 in the binary representation of N'. Let $\mathsf{Pos\_of\_Ind}(N', N) = w \cdot w'$ such that $|w'| = m$ (such w' exists thanks to Lemma B.13). We do a case analysis on N':

<u>Case $N' = 2^k$ for some k:</u> By definition of m, we deduce that $k = m$ and so $N' = 2^m$. Assume first that $\mathsf{VerifPoE}_{CT}(h, N, h', N', seq) = \mathsf{true}$. Thus $\mathsf{VerifPoP}_{CT}((h, N), h', (w, seq)) =$

true. By Lemma B.7, we can deduce that $(h', w) \in T$ and (w, seq) is the proof of presence of h' in T. But $(h', w) \in T$ implies that the subtree T' in T rooted by (h', w) is a ChronTree with root (h', ε) that we denote T'. Since $N' = 2^m$ and $\mathsf{Pos\_of\_Ind}(N', N) = w \cdot w'$ with $|w'| = m$, then by definition of a ChronTree, we deduce that T' is perfect and $\mathcal{S}(T') = [d_1, \ldots, d_{N'}]$. Therefore, we can directly deduce by Definition B.11 that seq is the proof of extension of T' into T. Thus the result holds.

Assume now that there exists T' of size N' with hash value h' and $\mathcal{S}(T') = [d_1, \ldots, d_{N'}]$ and seq is the proof of extension of T' into T. Note that T' is perfect thanks to $N' = 2^m$, hence we deduce by Lemma B.2 that $(h', w) \in T$. By Definition B.11, $(h', w) \in T$ implies that (w, seq) is the proof of presence of h' in T. Hence by Lemma B.7, we deduce that $\mathsf{VerifPoP}_{CT}((h, N), h', (w, seq)) = \mathsf{true}$. Therefore by Definition B.12, seq proves the extension of h' of size N' into h of size N.

<u>Case $N' \neq 2^{k'}$ for all k' and $w = \varepsilon$:</u> We show that this case is in fact impossible. $w = \varepsilon$ implies $|\mathsf{Pos\_of\_Ind}(N', N)| = m$. Let's consider the binary representation of N, *i.e.* $N = \sum_{i=1}^{q} 2^{a_i}$ where $a_1, \ldots, a_q$ is a strictly decreasing sequence of integers. By Lemma B.12, we deduce that:

— either $1 + \sum_{i=1}^{k-1} 2^{a_i} \leq N' \leq \sum_{i=1}^{k} 2^{a_i}$ for some $k \in \{1, \ldots, q-1\}$ and $m = a_k + k$
— otherwise $1 + \sum_{i=1}^{q-1} 2^{a_i} \leq N' \leq \sum_{i=1}^{q} 2^{a_i}$ and $m = a_q + q - 1$

But we already know that m is the smallest position of the bits 1 in the binary representation of N' and $a_1, \ldots, a_q$ is a strictly decreasing sequence of integers. Therefore, for some $k \in \{1, \ldots, q\}$, if $1 + \sum_{i=1}^{k-1} 2^{a_i} \leq N' \leq \sum_{i=1}^{k} 2^{a_i}$ then $m \leq a_k$. However, the result above showed us that when $k \in \{1, \ldots, q-1\}$, $m = a_k + k$. But $m \leq a_k$ implies that $k \leq 0$ which is a contradiction with $k \in \{1, \ldots, q-1\}$. At last, we also know that when $1 + \sum_{i=1}^{q-1} 2^{a_i} \leq N' \leq \sum_{i=1}^{q} 2^{a_i}$, we have that $m = a_q + q - 1$. With $m \leq a_q$, we deduce that $q \leq 1$ and so $q = 1$. In such a case, we deduce that $N = 2^{a_q} = 2^m = N'$ which is in contradiction with our hypothesis $N' < N$.

<u>Case $N' \neq 2^{k'}$ for all k' and $w \neq \varepsilon$:</u> Assume that $seq = [b_1, \ldots, b_k]$ for some k. Moreover, since T is a ChronTree, $h = \mathsf{h}(h_\ell, h_r)$ for some bitstrings h_ℓ, h_r and let T_ℓ (resp. T_r) be the subtree in T rooted by the left (resp. right) child of the root of T.

Assume first that $\mathsf{VerifPoE}_{CT}(h, N, h', N', seq) = \mathsf{true}$. By Definition B.12, we deduce that $|w| = k - 1$ (and so $k > 1$) and if we denote $w = a_k \cdot \ldots \cdot a_2$ and $S = [i_1, \ldots, i_p]$ is the increasing sequence of integer such that $\forall i \in S$, $a_i = r$; and $\forall i \in \{2, \ldots, k\} \smallsetminus S$, $a_i = \ell$ then we have that $\mathsf{VerifPoP}_{CT}((h, N), b_1, (w, [b_2, \ldots, b_k])) = \mathsf{true}$ and $\mathsf{VerifPoP}_{CT}((h', N'), b_1, (w_r, [b_{i_1}, \ldots, b_{i_p}])) = \mathsf{true}$ where $w_r \cdot w' = \mathsf{Pos\_of\_Ind}(N', N')$. By Lemma B.7, we obtain that $(w, [b_2, \ldots, b_k])$ is the proof of presence of b_1 in T. Hence by Definition B.11, we easily deduce that if there exists a ChronTree T' of size N', with hash value h' and $\mathcal{S}(T') = [d_1, \ldots, d_{N'}]$ then seq is the proof of extension of T' into T. It remains to the existence of such ChronTree T'. We do a case analysis on w:

Case $w = \ell \cdot w''$ for some w'': By Lemma B.7, $\mathsf{VerifPoP}_{CT}((h, N), b_1, (w, [b_2, \ldots, b_k])) = \mathsf{true}$ also implies that $(w, [b_2, \ldots, b_k])$ is the proof of presence of b_1 in T. But by Definition B.8, we have that $(w'', [b_2, \ldots, b_{k-1}])$ is the proof of presence of b_1 in T_ℓ, $(b_1, w_1) \in T_\ell$ and $h_r = b_k$. Since h_ℓ is the hash value of T_ℓ, and $2^{\mathsf{height}(T)-1}$ is its size, then we deduce by Lemma B.7 that $\mathsf{VerifPoP}_{CT}((h_\ell, 2^{\mathsf{height}(T)-1}), b_1, (w'', [b_2, \ldots, b_{k-1}])) = \mathsf{true}$. Moreover, since $w = \ell \cdot w''$, we can deduce that $a_k \notin S$, $N' \leq 2^{\mathsf{height}(T)-1}$. Thus $\mathsf{Pos\_of\_Ind}(N', N) = \ell \cdot \mathsf{Pos\_of\_Ind}(N', 2^{\mathsf{height}(T)-1})$ and so $\mathsf{Pos\_of\_Ind}(N', 2^{\mathsf{height}(T)-1}) = w'' \cdot w'$ with $|w'| = m$. In fact, since we assumed that $N' \neq 2^{k'}$ for all k', we have that $N' < 2^{\mathsf{height}(T)-1}$. At last, since we already showed that $\mathsf{VerifPoP}_{CT}((h', N'), b_1, (w_r, [b_{i_1}, \ldots, b_{i_p}])) = \mathsf{true}$, we obtain that

$\mathsf{VerifPoE}_{CT}(h_\ell, 2^{\mathsf{height}(T)-1}, h', N', [b_1, \dots, b_{k-1}]) = \mathsf{true}$. Hence we can apply our inductive hypothesis on $[b_1, \dots, b_{k-1}]$ and T_ℓ, and so we deduce that there exists a ChronTree T' of size N', with hash value h' and $\mathcal{S}(T') = [d_1, \dots, d_{N'}]$. Hence the result holds.

Case $w = r \cdot w''$ for some w'': By Lemma B.7, $\mathsf{VerifPoP}_{CT}((h, N), b_1, (w, [b_2, \dots, b_k])) = \mathsf{true}$ also implies that $(w, [b_2, \dots, b_k])$ is the proof of presence of b_1 in T. But by Definition B.8, we have that $(w'', [b_2, \dots, b_{k-1}])$ is the proof of presence of b_1 in T_r, $(b_1, w_1) \in T_r$ and $h_\ell = b_k$. Since h_r is the hash value of T_r, and $N - 2^{\mathsf{height}(T)-1}$ is its size, then we deduce by Lemma B.7 that $\mathsf{VerifPoP}_{CT}((h_r, N - 2^{\mathsf{height}(T)-1}), b_1, (w'', [b_2, \dots, b_{k-1}])) = \mathsf{true}$. Moreover, since $w = r \cdot w''$, we can deduce that $k = i_p$ and $N' > 2^{\mathsf{height}(T)-1}$. Thus $\mathsf{Pos\_of\_Ind}(N', N) = r \cdot \mathsf{Pos\_of\_Ind}(N' - 2^{\mathsf{height}(T)-1}, N - 2^{\mathsf{height}(T)-1})$ and $\mathsf{Pos\_of\_Ind}(N' - 2^{\mathsf{height}(T)-1}, N - 2^{\mathsf{height}(T)-1}) = w'' \cdot w'$. Furthermore, m is still the smallest position of the bits 1 in the binary representation of $N' - 2^{\mathsf{height}(T)-1}$.

On there other hand, we know that $\mathsf{Pos\_of\_Ind}(N', N') = w_r \cdot w'$ with $|w'| = m$. But Lemma B.11 and $2^{\mathsf{height}(T)} > N' > 2^{\mathsf{height}(T)-1}$ implies that $w_r \neq \varepsilon$. Moreover, we also have $\mathsf{Pos\_of\_Ind}(N', N') = r \cdot \mathsf{Pos\_of\_Ind}(N' - 2^{\mathsf{height}(T)-1}, N' - 2^{\mathsf{height}(T)-1})$. Thus by Definition B.9, $\mathsf{VerifPoP}_{CT}((h', N'), b_1, (w_r, [b_{i_1}, \dots, b_{i_p}])) = \mathsf{true}$ implies that there exist two bitstrings h'_ℓ, h'_r such that $h' = \mathsf{h}(h'_\ell, h'_r)$ and $\mathsf{VerifPoP}_{CT}((h'_r, N' - 2^{\mathsf{height}(T)-1}), b_1, (w'_r, [b_{i_1}, \dots, b_{i_{p-1}}])) = \mathsf{true}$ where $w_r = r \cdot w'_r$ and $h'_\ell = b_{i_p}$. But $\mathsf{Pos\_of\_Ind}(N' - 2^{\mathsf{height}(T)-1}, N' - 2^{\mathsf{height}(T)-1}) = w'_r \cdot w'$. Therefore we can deduce that that $\mathsf{VerifPoE}_{CT}(h_r, N - 2^{\mathsf{height}(T)-1}, h'_r, N' - 2^{\mathsf{height}(T)-1}, [b_1, \dots, b_{k-1}]) = \mathsf{true}$. By applying our inductive hypothesis on $[b_1, \dots, b_{k-1}]$ and T_r, we deduce that there exists a ChronTree T'_r of size $N' - 2^{\mathsf{height}(T)-1}$ with hash value h'_r and $\mathcal{S}(T'_r) = [d_{1+2^{\mathsf{height}(T)-1}}, \dots, d_{N'}]$.

At last, we already know that $h_\ell = b_k = b_{i_p} = h'_\ell$, $N' > 2^{\mathsf{height}(T)-1}$ and T_ℓ is perfect with $\mathsf{height}(T_\ell) = \mathsf{height}(T) - 1$. Thus, we deduce that $\mathsf{height}(T'_r) \leq \mathsf{height}(T_\ell)$. Thus we can create the tree T' rooted by $n_r = (\mathsf{h}(h_\ell, h'_r), \varepsilon)$ such that the subtree rooted by the left (resp. right) child of n_r is T_ℓ (resp. T'_r). In such a case, we obtain that T' is a ChronTree of size N' with hash value h' and such that $\mathcal{S}(T') = [d_1, \dots, d_{N'}]$. Hence the result holds.

Assume now that there exists a ChronTree T' of size N' with hash value h' and such that $\mathcal{S}(T') = [d_1, \dots, d_{N'}]$ and *seq* is the proof of extension of T' into T. Moreover, since T' is a ChronTree and $N' \neq 2^{k'}$ for all k', we deduce that $N' \neq 1$ and so $h' = \mathsf{h}(h'_\ell, h'_r)$ for some bitstrings h_ℓ, h_r and let T'_ℓ (resp. T'_r) be the subtree in T' rooted by the left (resp. right) child of the root of T'. Definition B.11 implies that $(w, [b_2, \dots, b_k])$ is the proof of presence of b_1 in T and $(b_1, w) \in T$. Thus by Lemma B.7, we deduce that $\mathsf{VerifPoP}_{CT}((h, N), b_1, (w, [b_2, \dots, b_k])) = \mathsf{true}$. We do a case analysis on w.

Case $w = \ell \cdot w''$ for some w'': We previously showed that in such a case, $N' < 2^{\mathsf{height}(T)-1}$. Moreover, since $w = \ell \cdot w''$ and $(b_1, w) \in T$, we deduce that $(b_1, w'') \in T_\ell$. By Definition B.8, $(w, [b_2, \dots, b_k])$ being the proof of presence of d_1 in T implies that $(w', [b_2, \dots, b_{k-1}])$ is the proof of presence of d_1 in T_ℓ. At last, since $N' < 2^{\mathsf{height}(T)-1} < N$, $\mathsf{Pos\_of\_Ind}(N', N) = \ell \cdot \mathsf{Pos\_of\_Ind}(N', 2^{\mathsf{height}(T)-1})$ and so $\mathsf{Pos\_of\_Ind}(N', 2^{\mathsf{height}(T)-1}) = w'' \cdot w'$ with $|w'| = m$. Therefore we deduce that $[b_1, \dots, b_{k-1}]$ is the proof of extension of T' into T_ℓ. By applying our inductive hypothesis on $[b_1, \dots, b_{k-1}]$, T_ℓ, T', we deduce that $\mathsf{VerifPoE}_{CT}(h_\ell, 2^{\mathsf{height}(T)-1}, h', N', [b_1, \dots, b_{k-1}]) = \mathsf{true}$. Hence by Definition B.12, if we denote $w'' = a_{k-1} \cdot \ldots \cdot a_2$ and $S = [i_1, \dots, i_p]$ is the increasing sequence of integer such that $\forall i \in S$, $a_i = r$; and $\forall i \in \{2, \dots, k-1\} \smallsetminus S$, $a_i = \ell$ then we have that $\mathsf{VerifPoP}_{CT}((h', N'), b_1, (w_r, [b_{i_1}, \dots, b_{i_p}])) = \mathsf{true}$ where $\mathsf{Pos\_of\_Ind}(N', N') = w_r \cdot w'$. Since $w = \ell \cdot w''$, then if we denote by $a_k = \ell$, we have that $\forall i \in \{2, \dots, k\} \smallsetminus S$, $a_i = \ell$ and $\mathsf{VerifPoP}_{CT}((h', N'), b_1, (w_r, [b_{i_1}, \dots, b_{i_p}])) = \mathsf{true}$. It allows us to conclude that $\mathsf{VerifPoE}_{CT}(h, N, h', N', seq) = \mathsf{true}$.

Case $w = r \cdot w''$ for some w'': We previously showed that in such a case, $N' > 2^{\mathsf{height}(T)-1}$. Thus since for all $i \in \{1,\dots,N'\}$, $d_i = d'_i$, we have that $T_\ell = T'_\ell$ thanks to Lemma B.2. Moreover, since $w = r \cdot w'$ and $(b_1, w) \in T$, we deduce that $(b_1, w') \in T_r$. By Definition B.8, $(w, [b_2,\dots,b_k])$ being the proof of presence of b_1 in T implies that $(w'', [b_2,\dots,b_{k-1}])$ is the proof of presence of b_1 in T_r with $b_k = h_\ell$. At last, since $2^{\mathsf{height}(T)-1} < N' < N \leq 2^{\mathsf{height}(T)}$, we deduce that m is still the smallest position of bits 1 in the binary representation of $N' - 2^{\mathsf{height}(T)-1}$, and that $\mathsf{Pos\_of\_Ind}(N', N) = r \cdot \mathsf{Pos\_of\_Ind}(N' - 2^{\mathsf{height}(T)-1}, N - 2^{\mathsf{height}(T)-1})$. Moreover, by Lemma B.11, we deduce that $\mathsf{Pos\_of\_Ind}(N' - 2^{\mathsf{height}(T)-1}, N - 2^{\mathsf{height}(T)-1}) = w'' \cdot w'$ and $w'' \neq \epsilon$. Hence $(b_1, w) \in T$ implies that $(b_1, w'') \in T_r$.

We do a case analysis on $N' - 2^{\mathsf{height}(T)-1}$:

— if $N' - 2^{\mathsf{height}(T)-1} = 2^k$ for some k, then we deduce that $b_1 = h'_r$ and $[b_2,\dots,b_{k-1}]$ is the proof of extension of T'_r into T_r. By our inductive hypothesis, we deduce that $\mathsf{VerifPoE}_{CT}(h_r, N - 2^{\mathsf{height}(T)-1}, h'_r, N' - 2^{\mathsf{height}(T)-1}, [b_2,\dots,b_{k-1}]) = \mathsf{true}$. Thus, we have that $\mathsf{VerifPoP}_{CT}((h_r, N - 2^{\mathsf{height}(T)-1}), h'_r, (w'', [b_2,\dots,b_{k-1}])) = \mathsf{true}$. Let's verify the different hypothesis of Definition B.12. Since $(w'', [b_2,\dots,b_{k-1}])$ is the proof of presence of b_1 in T_r, we deduce that $|w''| = k-2$ and so $|w| = k-1$. Let us denote $w = a_k \cdot \ldots \cdot a_2$. Moreover, let S be the strictly increasing sequence of integers $[i_1,\dots,i_p]$ such that $\forall i \in S$, $a_i = r$; and $\forall i \in \{2,\dots,k-1\} \smallsetminus S$, $a_i = \ell$. But by Lemma B.11, we know that $w'' = \ell \cdot \ldots \cdot \ell$ hence $w = r \cdot w''$ implies $S = [k]$. Moreover, by $N' - 2^{\mathsf{height}(T)-1} = 2^k$ and Lemma B.12, we deduce that $\mathsf{Pos\_of\_Ind}(N', N') = r \cdot \ldots \cdot r$ with $|\mathsf{Pos\_of\_Ind}(N', N')| = k + 2 - 1 = k + 1$. Hence $\mathsf{Pos\_of\_Ind}(N', N') = r \cdot w'$. At last, we know that $b_k = h_\ell = h'_\ell$ (since $T_\ell = T'_\ell$), $b_1 = h'_r$, $h' = \mathsf{h}(h'_\ell, h'_r)$ and $2^{\mathsf{height}(T)-1}$ is the size of T_ℓ. Hence by Definition B.9, we can deduce that $\mathsf{VerifPoP}_{CT}((h', N'), h'_r, (r, [h'_\ell])) = \mathsf{true}$, that is $\mathsf{VerifPoP}_{CT}((h', N'), b_1, (r, [b_k])) = \mathsf{true}$. Therefore we can conclude that $\mathsf{VerifPoE}_{CT}(h, N, h', N', seq) = \mathsf{true}$.

— Otherwise we deduce the $[b_1,\dots,b_{k-1}]$ is the proof of extension of T'_r into T_r. By our inductive hypothesis, we deduce that $\mathsf{VerifPoE}_{CT}(h_r, N - 2^{\mathsf{height}(T)-1}, h'_r, N' - 2^{\mathsf{height}(T)-1}, [b_1,\dots,b_{k-1}]) = \mathsf{true}$. Thus, if we denote w'' by $a_{k-1} \cdot \ldots \cdot a_2$ and $S = [i_1,\dots,i_p]$ is the increasing sequence of integer such that $\forall i \in S$, $a_i = r$; and $\forall i \in \{2,\dots,k-1\} \smallsetminus S$, $a_i = \ell$ then we have that $\mathsf{VerifPoP}_{CT}((h'_r, N' - 2^{\mathsf{height}(T)-1}), b_1, (w'_r, [b_{i_1},\dots,b_{i_p}])) = \mathsf{true}$ where $\mathsf{Pos\_of\_Ind}(N' - 2^{\mathsf{height}(T)-1}, N' - 2^{\mathsf{height}(T)-1}) = w'_r \cdot w'$ and $|w'| = m$. But we know that $h' = \mathsf{h}(h'_\ell, h'_r)$ where $h_\ell = h'_\ell = b_k$. Thus, by Definition B.9, we deduce that $\mathsf{VerifPoP}_{CT}((h', N'), b_1, (r \cdot w'_r, [b_{i_1},\dots,b_{i_p}, b_k])) = \mathsf{true}$. Note that $\mathsf{Pos\_of\_Ind}(N', N') = r \cdot \mathsf{Pos\_of\_Ind}(N' - 2^{\mathsf{height}(T)-1}, N' - 2^{\mathsf{height}(T)-1}) = (r \cdot w'_r) \cdot w'$. Moreover, if we denote $a_k = r$ and $S' = [i_1,\dots,i_p,k]$, then S' is the increasing sequence of integer such that $\forall i \in S'$, $a_i = r$; and $\forall i \in [2,k] \smallsetminus S'$, $a_i = \ell$. Hence we can conclude that $\mathsf{VerifPoE}_{CT}(h, N, h', N', seq) = \mathsf{true}$ with $seq = [b_1,\dots,b_k]$. $\square$

Definition B.13. Let X be a set of data. Let T be a ChronTree and let $(h, N) = \mathsf{digest}_c(T)$. Let dg be some bitstring and let seq be a sequence of bitstrings. We define $\mathsf{VerifPoE}_c(dg, (h, N), seq) = \exists h', N'$ s.t. $dg =^? (h', N') \wedge [(N <^? N' \wedge \mathsf{VerifPoE}_{CT}(h, N, h', N', seq)) \vee (N =^? N' \wedge h =^? h' \wedge seq =^? [\,])]$.

LEMMA B.15. *Let X be a set of data. Let T be a ChronTree and let $(h, N) = \mathsf{digest}_c(T)$. For all bitstring dg', we have that there exists a ChronTree T' such that $dg' = \mathsf{digest}_c(T')$ and $\mathsf{content}_c(T')$ is an initial subsequence of $\mathsf{content}_c(T)$ if, and only if, there exists a sequence of bitstring seq of size $O(\log(|\mathsf{content}_c(T)|))$ such that $\mathsf{VerifPoE}_c(dg', (h, N), seq) = \mathsf{true}$. Moreover, the computation time of $\mathsf{VerifPoE}_c$ is linear in the size of its input.*

PROOF. By definition, we have $\mathsf{content}_c(T) = \mathcal{S}(T)$. By Definition B.4, if $(h, N) = \mathsf{digest}_c(T)$ then h is the label of the root of T and N is the size of T, that is the number of leaves in T. Let's denote $\mathsf{content}_c(T) = [d_1, \dots, d_N]$. Moreover, let dg' be a bitstring.

Assume first that there exists a ChronTree T' such that $dg' = \mathsf{digest}_c(T')$ and $\mathsf{content}_c(T')$ is an initial subsequence of $\mathsf{content}_c(T)$. In such a case, there exists a bitstring h' and $N' \in \mathbb{N}$ such that $dg' = (h', N')$ where h' is the hash value of T' and N' is its size. Moreover, since $\mathsf{content}_c(T')$ is an initial subsequence of $\mathsf{content}_c(T)$, we can deduce that $N' \leq N$ and $\mathcal{S}(T') = [d_1, \dots, d_{N'}]$. We do a case analysis on N':
Case $N = N'$: In such a case $\mathsf{content}_c(T) = \mathsf{content}_c(T')$ and so by Lemma B.3, we deduce that $\mathsf{digest}_c(T) = \mathsf{digest}_c(T')$ and so $h = h'$. Hence by taking $seq = [\,]$, we can deduce that $\mathsf{VerifPoE}_c(dg', (h, N), seq) = \mathsf{true}$. Moreover, we trivially have that seq is of size $O(\log(N))$.
Case $N' < N$: By Definition B.13, we know that $\mathsf{Pos\_of\_Ind}(N', N) \geq m$ where m is the smallest position of bits 1 in the binary representation of N'. Thus, by Definition B.11, we know that there exists a sequence of bitstring seq such that seq is a proof of extension of T' into T. By Lemma B.14, we deduce that $\mathsf{VerifPoE}_{CT}(h, N, h', N', seq) = \mathsf{true}$ which allows us to conclude that $\mathsf{VerifPoE}_c(dg', (h, N), seq) = \mathsf{true}$. Note that by Lemma B.9, $\mathsf{size}(seq) \leq \max(\mathsf{size}_{\mathsf{data}}, \mathsf{size}_{\mathsf{hash}}) \times (\lceil \ln(N) \rceil + 1)$, that is $\mathsf{size}(seq) \in O(\log(N))$.

Assume now that there exists a sequence of bitstrings seq such that $\mathsf{VerifPoE}_c(dg', (h, N), seq) = \mathsf{true}$. It implies that $\exists h', N'$ such that $dg = (h', N')$ and one of the two following propositions holds:

— $N = N'$ and $h = h'$ and $seq = [\,]$: In such a case, by taking $T' = T$, the result trivially holds.
— $N < N'$ and $\mathsf{VerifPoE}_{CT}(h, N, h', N', seq) = \mathsf{true}$: In such a case, we apply Lemma B.14 and obtain that there exists a ChronTree T' with hash value h' and size N' such that T is an extension of T'. Hence we have that $dg' = \mathsf{digest}_c(T')$ and $\mathsf{content}_c(T') = \mathcal{S}(T')$ is an initial subsequence of $\mathsf{content}_c(T)$. Therefore the result holds.

Lastly, by Lemma B.10, we know that the computation time of $\mathsf{VerifPoE}_{CT}(h, N, h', N', seq)$ is $O(\ln(N))$. Since $\mathsf{size}(seq) \in O(\ln(N))$, h, h' are constant and $N' < N$ and $\mathsf{size}(N) \in O(\ln(N))$, we deduce that the computation time of $\mathsf{VerifPoE}_c$ is linear in the size of its input. □

B.4. Random checking

Thanks to Lemmas B.3, B.8 and B.15, we proved that ChronTree can be used to implement a chronological data structure. For its usage in DTKI, it remains to show that ChronTrees are randomly verifiable following Definition 3.2.

Definition B.14. We define the function $\mathsf{Rand}\exists_c$ such that for all bitstring dg, p, for all $N \in \mathbb{N}$, for all $n \in \{1, \dots, N\}$,

$$\mathsf{Rand}\exists_c(n, dg, N, p) = \exists h, d, seq \text{ s.t. } \left\{ \begin{array}{l} \quad p =^? (d, seq) \wedge dg =^? (h, N) \\ \wedge \mathsf{VerifPoP}_{CT}((h, N), d, (\mathsf{Pos\_of\_Ind}(n, N), seq)) \end{array} \right.$$

The boolean procedure consist of verifying a proof of presence of the n^{th} leaf in the ChronTree supposedly represented by the hash value h and the size N. We can prove that the existence of this ChronTree is ensure once all the proofs of presence are verified that is for every supposed leaves of the tree.

LEMMA B.16. *Let dg be a bitstring and let $N \in \mathbb{N}$. There exists a ChronTree T such that $dg = \mathsf{digest}_c(T)$ and $N = |\mathsf{content}_c(T)|$ if, and only if, for all $n \in \{1, \dots, N\}$, there exist a value p of size $O(\log(N))$ such that $\mathsf{Rand}\exists_c(n, dg, N, p) = \mathsf{true}$;*

PROOF. Let dg be a bitstring and let $N \in \mathbb{N}$. We first prove the right implication.

Assume that there there exists a ChronTree T such that $dg = \mathsf{digest}_c(T)$ and $N = |\mathsf{content}_c(T)|$. By definition, $dg = \mathsf{digest}_c(T)$ implies that there exists a bitstring h such that $dg = (h, N)$ and h is the hash value of T. Let's denote $\mathsf{content}_c(T)$ by $[d_1, \ldots, d_n]$. Let $n \in \{1, \ldots, N\}$. By Lemma B.4, $\mathsf{Pos\_of\_Ind}(n, N)$ is the position of the leaf labeled by d_n in T. By definition B.8, we know that there exists a sequence of bitstrings seq such that $(\mathsf{Pos\_of\_Ind}(n, N), seq)$ is the proof of presence of d_N in T. Therefore, by Lemma B.7, we can deduce that $\mathsf{VerifPoP}_{CT}((h, N), d_N, (\mathsf{Pos\_of\_Ind}(n, N), seq)) = \mathsf{true}$. With $dg = (h, N)$ and by denoting $p = (d_N, seq)$, this allows us to conclude that $\mathsf{Rand}\exists_c(n, dg, N, seq) = \mathsf{true}$. Note that by Lemma B.5, we know that $\mathsf{size}((\mathsf{Pos\_of\_Ind}(n, N), seq)) \in O(\ln(N))$ hence $\mathsf{size}(p) \in O(\log(N))$.

Let us now prove by induction on N that for all bistring dg, if for all $n \in \{1, \ldots, N\}$, there exist a value p such that $\mathsf{Rand}\exists_c(n, dg, N, p) = \mathsf{true}$ then there exists a ChronTree T such that $dg = \mathsf{digest}_c(T)$ and $N = |\mathsf{content}_c(T)|$.

Base case $N = 1$: In such a case, we have that there exists p such that $\mathsf{Rand}\exists_c(1, dg, 1, p) = \mathsf{true}$. Hence by Definition B.14, there exists h, d, seq such that $p = (d, seq)$, $dg = (h, N)$ and $\mathsf{VerifPoP}_{CT}((h, 1), d, (\mathsf{Pos\_of\_Ind}(1, 1), seq)) = \mathsf{true}$. Thus, if we define $T = \{(\varepsilon, h)\}$ then T is a ChronTree such that $\mathsf{content}_c(T) = [h]$. Hence $\mathsf{digest}_c(T) = (h, 1)$ and $1 = |\mathsf{content}_c(T)|$.

Inductive step $N > 1$: In such a case, there exists $k \in \mathbb{N}$ such that $2^{k-1} < N < 2^k$. We know that for all $n \in \{1, \ldots, N\}$, there exist a value p such that $\mathsf{Rand}\exists_c(n, dg, N, p) = \mathsf{true}$. Hence for all $n \in \{1, \ldots, N\}$, there exist h_n, d_n, seq_n such that $dg = (h_n, N)$ and $\mathsf{VerifPoP}_{CT}((h, N), d_n, (\mathsf{Pos\_of\_Ind}(n, N), seq_n)) = \mathsf{true}$. Therefore, there exists a bitstring h such that $dg = (h, N)$ and for all $n \in \{1, \ldots, N\}$, there exist d_n, seq_n such that $\mathsf{VerifPoP}_{CT}((h, N), d_n, (\mathsf{Pos\_of\_Ind}(n, N), seq_n)) = \mathsf{true}$.

Let us first focus on $n \in \{1, \ldots, 2^{k-1}\}$. By Definition B.5, we have that $\mathsf{Pos\_of\_Ind}(n, N), seq_n) = \ell \cdot \mathsf{Pos\_of\_Ind}(n, 2^{k-1})$. By Definition B.9, $\mathsf{VerifPoP}_{CT}((h, N), d_n, (\mathsf{Pos\_of\_Ind}(n, N), seq_n)) = \mathsf{true}$ implies that $\mathsf{CompPoP}(\mathsf{Pos\_of\_Ind}(n, N), seq_n, d_n) = h$ with $|seq_n| = |\mathsf{Pos\_of\_Ind}(n, N)|$. Therefore, if we denote $seq_n = [b_n^1, \ldots, b_n^m]$ then we have

$$\begin{aligned}\mathsf{CompPoP}(\mathsf{Pos\_of\_Ind}(n, N), seq_n, d_n) &= \mathsf{CompPoP}(\ell, [b_n^m], h') \\ &= \mathsf{h}(h', b_n^m)\end{aligned}$$

where $h' = \mathsf{CompPoP}(\mathsf{Pos\_of\_Ind}(n, 2^{k-1}), [b_n^1, \ldots, b_n^{m-1}], d_n)$. But since h is an injective function and this equality is true for all $n \in \{1, \ldots, 2^{k-1}\}$, we deduce that there exists h_ℓ and h_r such that $h = \mathsf{h}(h_\ell, h_r)$ and for all $n \in \{1, \ldots, 2^{k-1}\}$, there exists seq_n^ℓ, d_n such that $\mathsf{VerifPoP}_{CT}((h_\ell, 2^{k-1}), d_n, (\mathsf{Pos\_of\_Ind}(n, 2^{k-1}), seq_n^\ell)) = \mathsf{true}$. Therefore, we deduce that for all $n \in \{1, \ldots, 2^{k-1}\}$, there exists seq_n^ℓ, d_n such that $\mathsf{Rand}\exists_c(n, (h_\ell, 2^{k-1}), 2^{k-1}, (d_n, seq_n^\ell)) = \mathsf{true}$. By our inductive hypothesis, we can deduce that there exists a ChronTree T_ℓ such that $(h_\ell, 2^{k-1}) = \mathsf{digest}_c(T_\ell)$ and $2^{k-1} = |\mathsf{content}_c(T_\ell)|$.

Let us now focus on $n \in \{2^{k-1} + 1, \ldots, N\}$. By Definition B.5, we have that $\mathsf{Pos\_of\_Ind}(n, N), seq_n) = r \cdot \mathsf{Pos\_of\_Ind}(n - 2^{k-1}, N - 2^{k-1})$. By Definition B.9, $\mathsf{VerifPoP}_{CT}((h, N), d_n, (\mathsf{Pos\_of\_Ind}(n, N), seq_n)) = \mathsf{true}$ implies that $\mathsf{CompPoP}(\mathsf{Pos\_of\_Ind}(n, N), seq_n, d_n) = h$ with $|seq_n| = |\mathsf{Pos\_of\_Ind}(n, N)|$. Therefore, if we denote $seq_n = [b_n^1, \ldots, b_n^m]$ then we have

$$\begin{aligned}\mathsf{CompPoP}(\mathsf{Pos\_of\_Ind}(n, N), seq_n, d_n) &= \mathsf{CompPoP}(r, [b_n^m], h') \\ &= \mathsf{h}(b_n^m, h')\end{aligned}$$

where $h' = \mathsf{CompPoP}(\mathsf{Pos\_of\_Ind}(n - 2^{k-1}, N - 2^{k-1}), [b_n^1, \ldots, b_n^{m-1}], d_n)$. But h is an injective function and this equality is true for all $n \in \{2^{k-1} + 1, \ldots, N\}$. Moreover, we know that $h = \mathsf{h}(h_\ell, h_r)$. Thus we deduce that for all $n \in \{2^{k-1} + 1, \ldots, N\}$ there exists

seq_n^r, d_n such that $\mathsf{VerifPoP}_{CT}((h_r, N - 2^{k-1}), d_n, (\mathsf{Pos\_of\_Ind}(n - 2^{k-1}, N - 2^{k-1}), seq_n^r)) = \mathsf{true}$. It implies that for all $n' \in \{1, \ldots, N - 2^{k-1}\}$, there exists seq_n^r, d_n such that $\mathsf{VerifPoP}_{CT}((h_r, N - 2^{k-1}), d_n, (\mathsf{Pos\_of\_Ind}(n', N - 2^{k-1}), seq_n^r)) = \mathsf{true}$. Therefore, we deduce that for all $n \in \{1, \ldots, N - 2^{k-1}\}$, there exists seq_n^r, d_n such that $\mathsf{Rand}\exists_c(n, (h_r, N - 2^{k-1}), N - 2^{k-1}, (d_n, seq_n^r)) = \mathsf{true}$. By our inductive hypothesis, we can deduce that there exists a ChronTree T_r such that $(h_r, N - 2^{k-1}) = \mathsf{digest}_c(T_r)$ and $N - 2^{k-1} = |\mathsf{content}_c(T_r)|$.

Let us now build the binary tree T rooted by (ε, h) and such that the left (resp. right) child subtree of T is T_ℓ (resp. T_r). We know that T_ℓ is a ChronTree of size 2^{k-1} and hash value h_ℓ and T_r is a ChronTree of size $N - 2^{k-1}$ and hash value h_r where $2^{k-1} < N \leq 2^k$. Hence it results that T is a ChronTree of size N and hash value $h = \mathsf{h}(h_\ell, h_r)$. This allows us to conclude that $\mathsf{digest}_c(T) = (h, N)$ and $|\mathsf{content}_c(T)| = N$. □

Lastly, we have to verify the second property of a chronological data structure that is randomly verifiable. This is stated in the following lemma.

LEMMA B.17. *Let dg, dg' be two bistrings. Let $n \leq N < N'$ be three integers. If there exists a bitstring p_e such that $\mathsf{VerifPoE}_c(dg, dg', p_e) = \mathsf{true}$, $\mathsf{size}_c(dg) = N$ and $\mathsf{size}_c(dg') = N'$ then we have that there exists a bitstring p such that $\mathsf{Rand}\exists_c(n, dg, N, p) = \mathsf{true}$ if, and only if, there exists a bitstring p' such that $\mathsf{Rand}\exists_c(n, dg', N', p') = \mathsf{true}$.*

PROOF. Let dg, dg' be two bistrings. Let $n \leq N < N'$ be three integers. Let p_e be a bitstring such that $\mathsf{VerifPoE}_c(dg, dg', p_e) = \mathsf{true}$, $\mathsf{size}_c(dg) = N$ and $\mathsf{size}_c(dg') = N'$. We prove the result by induction on N:

Base case $N' = 1$: In such a case, we have $N = N' = 1$. Thus $\mathsf{VerifPoE}_c(dg, dg', p_e) = \mathsf{true}$ implies that $dg = dg'$ and so the result trivially holds.

Inductive step $N' > 1$: Otherwise, we know that there exists $k' \in \mathbb{N}$ such that $2^{k'-1} < N' \leq 2^{k'}$. We do a case analysis on n and N:

Case 1: $N \leq 2^{k'-1}$. By definition, $\mathsf{VerifPoE}_c(dg, dg', p_e) = \mathsf{true}$ implies that there exists h, h' such that $dg = (h, N)$ and $dg' = (h', N')$ such that $\mathsf{VerifPoE}_{CT}(h', N', h, N, p_e) = \mathsf{true}$. Thus by Definition B.12, p_e is a sequence of bitstrings $[b_1, \ldots, b_m]$ such that if we denote $\mathsf{Pos\_of\_Ind}(N, N') = w \cdot w'$ with $|w'|$ being the small position of bits 1 in the binary representation of N, then we have that:

— if $N = 2^k$ for some $k \in \mathbb{N}$ then $\mathsf{VerifPoP}_{CT}((h', N'), h, (w, p_e)) = \mathsf{true}$. Since $N < 2^{k'-1}$, we have that $\mathsf{Pos\_of\_Ind}(N, N') = \ell \cdot \mathsf{Pos\_of\_Ind}(N, 2^{k-1})$. Hence there exists w'' such that $w = \ell \cdot w''$ and so $\mathsf{Pos\_of\_Ind}(N, 2^{k-1}) = w'' \cdot w'$. Moreover, $\mathsf{VerifPoP}_{CT}((h', N'), h, (w, p_e)) = \mathsf{true}$ implies that $\mathsf{CompPoP}(w, p_e, h) = h'$, that is
$$\mathsf{CompPoP}(\ell, [b_m], \mathsf{CompPoP}(w'', [b_1, \ldots, b_{m-1}], h)) = h'$$
This allows us to deduce that $\mathsf{h}(\mathsf{CompPoP}(w'', [b_1, \ldots, b_{m-1}], h), b_m) = h'$. Therefore, there exists h'_ℓ, h'_r such that $h' = \mathsf{h}(h'_\ell, h'_r)$, $h'_\ell = \mathsf{CompPoP}(w'', [b_1, \ldots, b_{m-1}], h)$ and $h'_r = b_m$. This allows us to prove that $\mathsf{VerifPoE}_c((h, N), (h'_\ell, 2^{k'-1}), [b_1, \ldots, b_{m-1}]) = \mathsf{true}$.

— otherwise $|w| = m - 1$ and if we denote $w = a_m \cdot \ldots \cdot a_2$ and if $S = [i_1, \ldots, i_p]$ is the increasing sequence of integer such that $\forall i \in S$, $a_i = r$; and $\forall i \in \{2, \ldots, m\} \smallsetminus S$, $a_i = \ell$ then we have that $\mathsf{VerifPoP}_{CT}((h', N'), b_1, (w, [b_2, \ldots, b_m])) = \mathsf{true}$ and $\mathsf{VerifPoP}_{CT}((h, N), b_1, (w_r, [b_{i_1}, \ldots, b_{i_p}])) = \mathsf{true}$ where $\mathsf{Pos\_of\_Ind}(N, N) = w_r \cdot w'$. Since $N < 2^{k'-1}$, we have that $\mathsf{Pos\_of\_Ind}(N, N') = \ell \cdot \mathsf{Pos\_of\_Ind}(N, 2^{k-1})$. Hence there exists w'' such that $w = \ell \cdot w''$ and so $\mathsf{Pos\_of\_Ind}(N, 2^{k-1}) = w'' \cdot w'$. Moreover, $\mathsf{VerifPoP}_{CT}((h', N'), b_1, (w, [b_2, \ldots, b_m])) = \mathsf{true}$ implies that $\mathsf{CompPoP}(w, [b_2, \ldots, b_m], b_1) = h'$, that is
$$\mathsf{CompPoP}(\ell, [b_m], \mathsf{CompPoP}(w'', [b_2, \ldots, b_{m-1}], b_1)) = h'$$

This allows us to deduce that $\mathsf{h}(\mathsf{CompPoP}(w'', [b_2, \dots, b_{m-1}], b_1), b_m) = h'$. Therefore, there exists h'_ℓ, h'_r such that $h' = \mathsf{h}(h'_\ell, h'_r)$, $h'_\ell = \mathsf{CompPoP}(w'', [b_2, \dots, b_{m-1}], b_1)$ and $h'_r = b_m$. This allows us to prove that $\mathsf{VerifPoE}_c((h, N), (h'_\ell, 2^{k'-1}), [b_1, \dots, b_{m-1}]) = \mathsf{true}$.

In both cases, we showed that there exist h'_ℓ, h'_r and a sequence of bistrings seq such that $h' = \mathsf{h}(h'_\ell, h'_r)$, $p_e = seq :: h'_r$ and $\mathsf{VerifPoE}_c((h, N), (h'_\ell, 2^{k'-1}), seq) = \mathsf{true}$. By our inductive hypothesis, we can deduce that that there exists a bitstring p such that $\mathsf{Rand}\exists_c(n, (h, N), N, p) = \mathsf{true}$ if, and only if, there exists a bitstring p' such that $\mathsf{Rand}\exists_c(n, (h'_\ell, 2^{k'-1}), 2^{k'-1}, p') = \mathsf{true}$. But $\mathsf{Rand}\exists_c(n, (h'_\ell, 2^{k'-1}), 2^{k'-1}, p') = \mathsf{true}$ is equivalent to there exists d_n, s_n such that $p' = (d_n, s_n)$ and $\mathsf{VerifPoP}_{CT}((h'_\ell, 2^{k'-1}), d_n, (\mathsf{Pos\_of\_Ind}(n, 2^{k'-1}), s_n)) = \mathsf{true}$. But $\ell \cdot \mathsf{Pos\_of\_Ind}(n, 2^{k'-1}) = \mathsf{Pos\_of\_Ind}(n, N')$ and $h' = \mathsf{h}(h'_\ell, h'_r)$. Therefore, we deduce that

$$\begin{aligned} & \mathsf{VerifPoP}_{CT}((h'_\ell, 2^{k'-1}), d_n, (\mathsf{Pos\_of\_Ind}(n, 2^{k'-1}), s_n)) \\ = \; & \mathsf{VerifPoP}_{CT}((h', N'), d_n, (\mathsf{Pos\_of\_Ind}(n, N'), s_n :: h'_r)) \\ = \; & \mathsf{true} \end{aligned}$$

It allows us to conclude that $\mathsf{Rand}\exists_c(n, (h'_\ell, 2^{k'-1}), 2^{k'-1}, p') = \mathsf{true}$ is equivalent to $\mathsf{Rand}\exists_c(n, (h', N'), N', (d_n, s_n :: h'_r)) = \mathsf{true}$.

Case 2: $n < 2^{k'-1} + 1 \leq N < N'$. In such a case, we know that $N \neq 2^k$ for all $k \in \mathbb{N}$. By definition, $\mathsf{VerifPoE}_c(dg, dg', p_e) = \mathsf{true}$ implies that there exists h, h' such that $dg = (h, N)$ and $dg' = (h', N')$ such that $\mathsf{VerifPoE}_{CT}(h', N', h, N, p_e) = \mathsf{true}$. Thus by Definition B.12, p_e is a sequence of bitstrings $[b_1, \dots, b_m]$ such that if we denote $\mathsf{Pos\_of\_Ind}(N, N') = w \cdot w'$ with $|w'|$ being the small position of bits 1 in the binary representation of N, then we have that $|w| = m - 1$. Moreover, if we denote $w = a_m \cdot \ldots \cdot a_2$ and if $S = [i_1, \dots, i_p]$ is the increasing sequence of integer such that $\forall i \in S$, $a_i = r$; and $\forall i \in \{2, \dots, m\} \smallsetminus S$, $a_i = \ell$ then we have that $\mathsf{VerifPoP}_{CT}((h', N'), b_1, (w, [b_2, \dots, b_m])) = \mathsf{true}$ and $\mathsf{VerifPoP}_{CT}((h, N), b_1, (w_r, [b_{i_1}, \dots, b_{i_p}])) = \mathsf{true}$ where $\mathsf{Pos\_of\_Ind}(N, N) = w_r \cdot w'$.

Since $2^{k'-1} + 1 \leq N < N'$, we have that $\mathsf{Pos\_of\_Ind}(N, N') = r \cdot \mathsf{Pos\_of\_Ind}(N - 2^{k-1}, N' - 2^{k-1})$. Hence there exists w'' such that $w = r \cdot w''$ and so $\mathsf{Pos\_of\_Ind}(N - 2^{k-1}, N' - 2^{k-1}) = w'' \cdot w'$. Moreover, $\mathsf{VerifPoP}_{CT}((h', N'), b_1, (w, [b_2, \dots, b_m])) = \mathsf{true}$ implies that $\mathsf{CompPoP}(w, [b_2, \dots, b_m], b_1) = h'$, that is

$$\mathsf{CompPoP}(r, [b_m], \mathsf{CompPoP}(w'', [b_2, \dots, b_{m-1}], b_1)) = h'$$

This allows us to deduce that $\mathsf{h}(b_m, \mathsf{CompPoP}(w'', [b_2, \dots, b_{m-1}], b_1)) = h'$. Therefore, there exists h'_ℓ, h'_r such that $h' = \mathsf{h}(h'_\ell, h'_r)$, $h'_\ell = b_m$ and $h'_r = \mathsf{CompPoP}(w'', [b_2, \dots, b_{m-1}], b_1)$.

$w = r \cdot w''$ also implies that $i_p = m$. Moreover, since $N > 2^{k'-1}$, we have that $\mathsf{Pos\_of\_Ind}(N, N) = r \cdot \mathsf{Pos\_of\_Ind}(N - 2^{k-1}, N - 2^{k-1})$. Therefore, there exists w'_r such that $w_r = r \cdot w'_r$ and $\mathsf{Pos\_of\_Ind}(N, N) = w'_r \cdot w'$. This allow us to deduce that $\mathsf{VerifPoP}_{CT}((h, N), b_1, (w_r, [b_{i_1}, \dots, b_{i_p}])) = \mathsf{true}$ implies $\mathsf{CompPoP}(w_r, [b_{i_1}, \dots, b_{i_p}], b_1) = h$, that is

$$\mathsf{CompPoP}(r, [b_m], \mathsf{CompPoP}(w'_r, [b_{i_1}, \dots, b_{i_p - 1}], b_1)) = h$$

This allows us to deduce that $\mathsf{h}(b_m, \mathsf{CompPoP}(w'_r, [b_{i_1}, \dots, b_{i_p-1}], b_1)) = h$. Therefore, there exists h_ℓ, h_r such that $h = \mathsf{h}(h_\ell, h_r)$, $h_\ell = b_m = h'_\ell$ and $h_r = \mathsf{CompPoP}(w'_r, [b_{i_1}, \dots, b_{i_p-1}], b_1)$. It means that $\mathsf{VerifPoE}_c((h_r, N - 2^{k-1}), (h'_r, N' - 2^{k'-1}), [b_1, \dots, b_{m-1}]) = \mathsf{true}$.

But $n < 2^{k'-1} + 1 \leq N < N'$ implies that $\mathsf{Pos\_of\_Ind}(n, N) = \mathsf{Pos\_of\_Ind}(n, N') = \ell \cdot \mathsf{Pos\_of\_Ind}(n, 2^{k'-1})$. Moreover, we already prove that $h = \mathsf{h}(h_\ell, h_r)$ and $h' = \mathsf{h}(h_\ell, h_r)$. Thus $\mathsf{Rand}\exists_c(n, (h, N), N, p) = \mathsf{true}$ is equivalent to there exists d_n, s_n such that $p = (d_n, s_n)$

and $\mathsf{VerifPoP}_{CT}((h,N), d_n, (\mathsf{Pos\_of\_Ind}(n,N), s_n)) = \mathsf{true}$. Thus we deduce that there exists seq such that $s_n = seq :: h_r$ and $\mathsf{VerifPoP}_{CT}((h_\ell, 2^{k'-1}), d_n, (\mathsf{Pos\_of\_Ind}(n, 2^{k'-1}), seq)) = \mathsf{true}$. But it also implies that $\mathsf{VerifPoP}_{CT}((h',N'), d_n, (\mathsf{Pos\_of\_Ind}(n,N'), seq :: h'_r)) = \mathsf{true}$. The other implication being similar, we deduce that there exists p such that $\mathsf{Rand}\exists_c(n,(h,N),N,p) = \mathsf{true}$ is equivalent to there exists p' such that $\mathsf{Rand}\exists_c(n,(h',N'),N',p') = \mathsf{true}$. Hence the result holds.

<u>Case 3: $2^{k'-1}+1 \le n \le N < N'$</u> For this case, by definition, $\mathsf{VerifPoE}_c(dg, dg', p_e) = \mathsf{true}$ implies that there exists h, h' such that $dg = (h,N)$ and $dg' = (h',N')$ such that $\mathsf{VerifPoE}_{CT}(h',N',h,N,p_e) = \mathsf{true}$. Since $2^{k'-1} < N < N' \le 2^{k'}$, we already showed in Case 2 that there exists h_ℓ, h_r, h'_r and a sequence of bitstrings seq such that $h = \mathsf{h}(h_\ell, h_r)$, $h' = \mathsf{h}(h_\ell, h'_r)$, $p_e = seq :: h_\ell$ and $\mathsf{VerifPoE}_c((h_r, N-2^{k-1}), (h'_r, N'-2^{k'-1}), seq) = \mathsf{true}$.

By our inductive hypothesis, we deduce that for all $n' \in \{1,\dots,N-2^{k'-1}\}$, there exists a bitstring p such that $\mathsf{Rand}\exists_c(n', (h_r, N-2^{k-1}), N-2^{k-1}, p) = \mathsf{true}$ if, and only if, there exists a bitstring p' such that $\mathsf{Rand}\exists_c(n', (h'_r, N'-2^{k'-1}), N'-2^{k'-1}, p') = \mathsf{true}$. Note that $2^{k'-1}+1 \le n \le N$. Thus $\mathsf{Rand}\exists_c(n-2^{k'-1}, (h_r, N-2^{k'-1}), N-2^{k'-1}, p) = \mathsf{true}$ is equivalent to there exists d_n, s_n such that $p = (d_n, s_n)$ and $\mathsf{VerifPoP}_{CT}((h_r, N-2^{k'-1}), d_n, (\mathsf{Pos\_of\_Ind}(n-2^{k'-1}, N-2^{k'-1}), s_n)) = \mathsf{true}$. But $r \cdot \mathsf{Pos\_of\_Ind}(n-2^{k'-1}, N-2^{k'-1}) = \mathsf{Pos\_of\_Ind}(n,N)$ and $h' = \mathsf{h}(h'_\ell, h'_r)$. Therefore, we deduce that

$$\begin{aligned}
& \mathsf{VerifPoP}_{CT}((h_r, N-2^{k'-1}), d_n, (\mathsf{Pos\_of\_Ind}(n-2^{k'-1}, N-2^{k'-1}), s_n)) \\
= \; & \mathsf{VerifPoP}_{CT}((h,N), d_n, (\mathsf{Pos\_of\_Ind}(n,N), s_n :: h_r)) \\
= \; & \mathsf{true}
\end{aligned}$$

Thus, we deduce that $\mathsf{Rand}\exists_c(n-2^{k'-1}, (h_r, N-2^{k-1}), N-2^{k-1}, (d_n, s_n)) = \mathsf{true}$ is equivalent to $\mathsf{Rand}\exists_c(n, (h,N), N, (d_n, s_n :: h_r)) = \mathsf{true}$.

Similarly, $\mathsf{Rand}\exists_c(n-2^{k'-1}, (h'_r, N'-2^{k'-1}), N'-2^{k'-1}, p') = \mathsf{true}$ is equivalent to there exists d'_n, s'_n such that $p' = (d'_n, s'_n)$ and $\mathsf{VerifPoP}_{CT}((h'_r, N'-2^{k'-1}), d'_n, (\mathsf{Pos\_of\_Ind}(n-2^{k'-1}, N'-2^{k'-1}), s'_n)) = \mathsf{true}$. Since $r \cdot \mathsf{Pos\_of\_Ind}(n-2^{k'-1}, N'-2^{k'-1}) = \mathsf{Pos\_of\_Ind}(n, N')$, we deduce that

$$\begin{aligned}
& \mathsf{VerifPoP}_{CT}((h'_r, N'-2^{k'-1}), d_n, (\mathsf{Pos\_of\_Ind}(n-2^{k'-1}, N'-2^{k'-1}), s'_n)) \\
= \; & \mathsf{VerifPoP}_{CT}((h',N'), d'_n, (\mathsf{Pos\_of\_Ind}(n,N'), s'_n :: h'_r)) \\
= \; & \mathsf{true}
\end{aligned}$$

Thus, we deduce that $\mathsf{Rand}\exists_c(n-2^{k'-1}, (h'_r, N'-2^{k-1}), N'-2^{k-1}, (d'_n, s'_n)) = \mathsf{true}$ is equivalent to $\mathsf{Rand}\exists_c(n, (h',N'), N,')(d'_n, s'_n :: h'_r) = \mathsf{true}$. Hence the result holds. $\square$

C. IMPLEMENTATION OF THE ORDERED DATA STRUCTURE

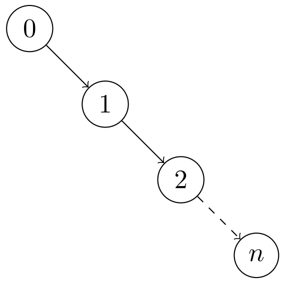
Contrary to a chronological data structure, an ordered data structure is much more malleable to change since it allows one to prove addition, deletion, presence and absence of an element in the data structure. Naturally, we cannot use again the ChronTree to implement the ordered data structure. However, we will keep the core of it, that is the usage of hash tree, to provide cryptographic proofs. On the other hand, an ordered data structure is a data structure over a partially ordered set of elements. Thus, we will combine hash tree with a classical tree structure, that is the AVL tree to model the ordered data structure.

C.1. AVL hash tree

Definition C.1. A *binary search tree* T over a set D partially ordered by the relation $\mathcal{R}$ is a binary tree over D such that for all $n, n' \in T$, if b and b' are the respective labels of n and n' then $b \,\mathcal{R}\, b'$ if, and only if, $n \,\mathcal{O}_T\, n'$.

A binary search tree provides operations such as search and addition of an element, whose complexity is linear in the height of the tree. Then, if the tree is equilibrate, that is complete, the height of the tree is logarithmic in the number of nodes in the tree. However, depending on the order in which the elements are added in the tree, the tree may not always be complete as shown in Example C.1.

Example C.1. Consider the binary search tree over $\mathbb{N}$ naturally ordered, described as follows.



We have that the height of the tree is linear in the number of nodes and not logarithmic as requested in the specification of ordered data structure. $\diamond$

To ensure that the height of the tree is always logarithmic in the number of node, we consider a classical tree structure, that is *AVL tree*.

Definition C.2. An AVL tree T over a set D ordered by $\mathcal{R}$ is a binary search tree over D such that for all nodes n, the heights of the two child subtrees of n differ by at most one.

Note that an AVL tree is not necessarily complete but its height is always logarithmic in the number of nodes, as stated in the following lemma.

LEMMA C.1. *Let T be an AVL tree, N be its number of node and H its height. We have that $H \in \log(N)$.*

PROOF. Let us compute the AVL tree T_H that has the smallest number of node for a given height H. Let us denote U_n the number of node of T_H. For $H = 0$, T_H is unique and is composed of only the root, which implies $U_0 = 1$.

Consider $H > 0$. Let us denote by T_ℓ (resp. T_r) the left (resp. right) child subtree of the root of T_H. By definition of an AVL tree, we know that $|\mathsf{height}(T_\ell) - \mathsf{height}(T_r)| \leq 1$. Since we assume that T_H has the smallest number of node, we have that $|\mathsf{height}(T_\ell) - \mathsf{height}(T_r)| = 1$ and we can consider w.r.t. that $\mathsf{height}(T_\ell) - \mathsf{height}(T_r) = 1$. It implies that $\mathsf{height}(T_r) + 2 = \mathsf{height}(T_\ell) + 1 = \mathsf{height}(T_H)$. At last, since the number of node in T_H is the number of node in T_ℓ plus the number of node in T_r plus one, we obtain that $U_H = U_{H-1} + U_{H-2} + 1$. Thus, we deduce that $U_H > 2 \times U_{H-2}$ and so $U_H > 2^{\frac{H}{2}}$. It implies that $\frac{2 \times \ln(U_H)}{\ln(2)} > H$. Therefore, we can conclude that $H \in \log(U_H)$. $\square$

Given a node n in a tree T, we denote by $\mathsf{fctH}_T(n)$, called *height factor of n in T*, the difference of height between the left and right child subtree of n in T, *i.e.* $\mathsf{fctH}_T(n) = H_\ell - H_r$ where H_ℓ (resp. H_r) is the height of the left (resp. right) child subtree of n in T when it exists, -1 otherwise.

Example C.2. Let D a set and consider an AVL tree T, represented as below, where $d_1, \ldots, d_7 \in D$ and for all $i \in \{1, \ldots, 6\}$, $d_i \,\mathcal{R}\, d_{i+1}$.

$$\begin{tikzcd}
 & & d_3 \arrow[dll] \arrow[drrr] & & & & \\
d_1 \arrow[dr] & & & & & d_6 \arrow[dl] \arrow[dr] & \\
 & d_2 & & & d_5 \arrow[dl] & & d_7 \\
 & & & d_4 & & &
\end{tikzcd}$$

Consider the three nodes $n = (d_2, \ell \cdot r)$, $n' = (d_3, \varepsilon)$ and $n'' = (d_5, r \cdot \ell)$. We have that $\mathsf{fctH}_T(n) = 0$, $\mathsf{fctH}_T(n') = -1$ and $\mathsf{fctH}_T(n'') = 1$. $\diamond$

We now describe how we combine hash trees and AVL hash trees to obtain the tree structure that implements the ordered data structure.

Definition C.3. An AVL hash tree T over the set D partially ordered by $\mathcal{R}$ is an AVL tree over the set $D \times \mathcal{H}$ ordered by $\mathcal{R}'$ such that:

— for all $(d, h), (d', h') \in D \times \mathcal{H}$, $(d, h)\, \mathcal{R}' (d', h')$ if and only if $d\, \mathcal{R}\, d'$;
— for all nodes $n \in T$, n is labeled with $(d, \mathsf{h}(d, H, f, h_\ell, h_r))$ where $d \in D$, H is the height of the subtree rooted by n, $f = \mathsf{fctH}_T(n)$ and (d_ℓ, h_ℓ) (resp. (d_r, h_r)) is the label of n's left child (resp. right child) if it exists, or $h_\ell = \mathsf{null}$ (resp. $h_r = \mathsf{null}$) otherwise.

The set of data stored in T is the set $\mathsf{Data}(T) = \{d \mid (d, h) \in T\}$.

The AVL hash tree preserves the idea of an hash tree by requiring that the label of a node contains the hash of the labels of its childs. Moreover, it preserves the nice properties of a binary search tree by ordering the labels with only the data stored in the tree, *i.e.* the elements of D, and thus disregarding the hash values. However, to ensure that the tree is an AVL tree, we require additional informations to be hashed, that is the height of the subtree and its height factor in the tree.

Consider a node n labeled by (d, h), we say that d is the data stored in n and h is the hash value of n.

Example C.3. Consider a set D partially ordered by $\mathcal{R}$. Let $d_1, d_2, d_3, d_4, d_5, d_6, d_7 \in D$ such that for all $i \in \{1, \ldots, 6\}$, $d_i\, \mathcal{R}\, d_{i+1}$. Consider the following AVL tree T where $h_i = \mathsf{h}(d_i, 0, 0, \mathsf{null}, \mathsf{null})$ for all $i = \{2, 4, 7\}$.

$$\begin{tikzcd}
 & & \boxed{d_3, \mathsf{h}(d_3, 3, -1, \mathsf{h}(d_1, 1, -1, \mathsf{null}, h_2), \mathsf{h}(d_6, 2, 1, \mathsf{h}(d_5, 1, 1, h_4, \mathsf{null}), h_7))} \arrow[dll] \arrow[drrr] & & & & \\
\boxed{d_1, \mathsf{h}(d_1, 1, -1, \mathsf{null}, h_2)} \arrow[dr] & & & & & \boxed{d_6, \mathsf{h}(d_6, 2, 1, \mathsf{h}(d_5, 1, 1, h_4, \mathsf{null}), h_7)} \arrow[dl] \arrow[dr] & \\
 & \boxed{d_2, h_2} & & & \boxed{d_5, \mathsf{h}(d_5, 1, 1, h_4, \mathsf{null})} \arrow[dl] & & \boxed{d_7, h_7} \\
 & & & \boxed{d_4, h_4} & & &
\end{tikzcd}$$

The tree T is an AVL hash tree. $\diamond$

Note that the AVL hash tree in Example C.3 is the AVL hash tree corresponding to the AVL tree in Example C.2. In fact, we show in the next lemma that there is a bijection between the AVL and AVL hash trees.

LEMMA C.2. *Let D be a set partially ordered by $\mathcal{R}$. There exists a bijection* $\mathsf{to_h}$ *from AVL trees over D to AVL hash trees over D such that for all AVL tree T, for all $d \in D$, for all $w \in \mathcal{A}^*$, $(d, w) \in T$ if, and only if, there exists $h \in \mathcal{H}$ such that $((d, h), w) \in \mathsf{to_h}(T)$.*

PROOF. The function $\mathsf{to_h}$ directly following the definition of an AVL hash tree: Given an AVL tree T, $|T| = |\mathsf{to_h}(T)|$ and for all $(d, w) \in T$, we have that $n = ((d, \mathsf{h}(d, f, H, h_\ell, h_r)), w) \in \mathsf{to_h}T$ where H is the height of the subtree rooted by n, $f = \mathsf{fctH}_{\mathsf{to_h}T}(n)$ and (d_ℓ, h_ℓ) (resp. (d_r, h_r)) is the label of n's left child (resp. right child) if it exists, or $h_\ell = \mathsf{null}$ (resp. $h_r = \mathsf{null}$) otherwise.

It remains to show that $\mathsf{to_h}$ is injective, that is for all AVL trees T and T', $\mathsf{to_h}(T) = \mathsf{to_h}(T')$ implies that $T = T'$. The function $\mathsf{to_h}$ does not modify the structure of the tree, thus $\mathsf{to_h}(T) = \mathsf{to_h}(T')$ implies that T and T' have the same structure, they may only differ by their labels. But we also know that for all nodes $n \in \mathsf{to_h}(T)$, there exists $d \in D$, $w \in \mathcal{A}^*$, $h \in \mathcal{H}$ such that $((d, h), w) \in \mathsf{to_h}(T)$. But it implies that $(d, w) \in T$. With the same reasoning on T', we deduce that $T = T'$. □

C.2. Proof of presence

As for the ChronTree, we can generate a proof that allows one to verify that some data are in the AVL hash tree.

Definition C.4 (*Proof of presence*). Let D be a set partially ordered by $\mathcal{R}$. Let T be an AVL hash tree and $d \in D$. The *proof of presence of d in T* exists if there is a node n_0 in T labeled by $(d, \mathsf{h}(d, H, f, h_\ell, h_r))$ for some $H, f \in \mathbb{N}$ and $h_\ell, h_r \in \mathcal{H}$; and if $n_k, \ldots, n_0$ is the path to n_0 in T then the proof of presence of d in T is defined as the tuple $(H, f, h_\ell, h_r, [(d_1, H_1, f_1), \ldots, (d_k, H_k, f_k)], [h_1, \ldots, h_k])$ such that for all $i \in \{1, \ldots, k\}$,

— n_i is labeled by $(d_i, \mathsf{h}(d_i, H_i, f_i, h, h'))$ for some $h, h' \in \mathcal{H}$; and
— h_i is the hash value of $\mathsf{Sib}_T(n_{i-1})$ if it exists else $h_i = \mathsf{null}$.

In the proof of presence for AVL hash tree, instead of putting the position of the node n that stores d (as for ChronTree), we put a sequence of data which is stored in the nodes on the path from the root to n, as well as the informations specific to the AVL, that are the heights and height factors of each node in the path.

In fact, the path (so the position) from the root to node n can be computed from the sequence thanks to the structure of binary search tree. This operation also ensures that this portion of the AVL hash tree indeed satisfies the properties of a binary search tree i.e. the nodes on this path are ordered as what the binary search tree specifies, and the properties of the AVL tree, *i.e.* the relation between heights and height factors on this are correct.

Example C.4. Consider the AVL hash tree of Example C.3 that we denote T. The proof of presence of d_5 in T is the tuple $(1, 1, h_4, \mathsf{null}, seq_d, seq_h)$ where:

— $seq_d = [(d_6, 2, 1), (d_3, 3, -1)]$
— $seq_h = [\mathsf{h}(d_7, \mathsf{null}, \mathsf{null}), \mathsf{h}(d_1, 1, -1, \mathsf{null}, \mathsf{h}(d_2, \mathsf{null}, \mathsf{null}))]$

The proof of presence $(h_\ell, h_r, seq_d, seq_h)$ is graphically represented in Figure 9. The data in the light gray nodes the elements in seq_d whereas the hash values of the dark gray nodes are the elements of seq_h. ◇

Like in a ChronTree, verifying the proof of presence of some data in an AVL hash tree T mainly consists of reconstructing the hash value of the root of T.

Definition C.5 (*Verification of a proof of presence*). Let D be a set partially ordered by the relation $\mathcal{R}$. Let $d \in D$, $h \in \mathcal{H}$ and a tuple $pr = (H, f, h_\ell, h_r, seq_d, seq_h)$. The verification that pr proves the presence of d in h, denoted $\mathsf{VerifPoP}_{AVL}(h, d, pr)$, is defined as follows.

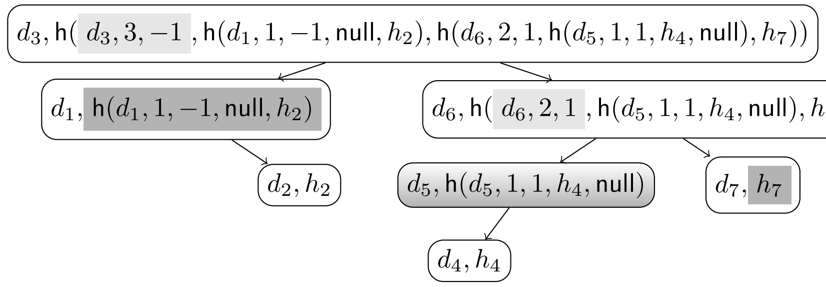

$d_3, \mathsf{h}(d_3, 3, -1, \mathsf{h}(d_1, 1, -1, \mathsf{null}, h_2), \mathsf{h}(d_6, 2, 1, \mathsf{h}(d_5, 1, 1, h_4, \mathsf{null}), h_7))$
$d_1, \mathsf{h}(d_1, 1, -1, \mathsf{null}, h_2)$
$d_6, \mathsf{h}(d_6, 2, 1, \mathsf{h}(d_5, 1, 1, h_4, \mathsf{null}), h_7)$
d_2, h_2
$d_5, \mathsf{h}(d_5, 1, 1, h_4, \mathsf{null})$
d_7, h_7
d_4, h_4

Fig. 9. Graphical representation of the proof of presence of d_5 in the AVL hash tree T.

We have that $\mathsf{VerifPoP}_{AVL}(h, d, pr) = \mathsf{true}$ if, and only if seq_d, seq_h are two sequences of bitstrings, $|seq_d| = |seq_h|$ and $h = \mathsf{CompPoP}_{AVL}(seq_d, seq_h, \mathsf{h}(d, H, f, h_\ell, h_r), d)$ where $\mathsf{CompPoP}_{AVL}(seq_d, seq_h, h, d)$ is defined recursively as follows:

— $\mathsf{CompPoP}_{AVL}([], [], h, d) = h$
— if $seq_d = (d_1, H_1, f_1) :: seq'_d$, $seq_h = h_1 :: seq'_h$, and $d \mathcal{R} d_1$ then

$$\mathsf{CompPoP}_{AVL}(seq_d, seq_h, h, d) = \mathsf{CompPoP}_{AVL}(seq'_d, seq'_h, \mathsf{h}(d_1, H_1, f_1, h, h_1), d)$$

— if $seq_d = (d_1, H_1, f_1) :: seq'_d$, $seq_h = h_1 :: seq'_h$, and $d_1 \mathcal{R} d$ then

$$\mathsf{CompPoP}_{AVL}(seq_d, seq_h, h, d) = \mathsf{CompPoP}_{AVL}(seq'_q, seq'_h, \mathsf{h}(d_1, H_1, f_1, h_1, h), d_1)$$

Compared to the verification of presence of some data in a ChonTree, the position of the node is determined with the relation between d and data $d_1, d_2, \ldots$ stored in the nodes on the path to the

We can now prove the soundness and completeness of the verification of a proof of presence as stated in the following lemma.

LEMMA C.3. *Let D be a set partially ordered by the relation $\mathcal{R}$. Let T be an AVL hash tree whose hash value is h. Let $d \in D$ and a tuple $(H, f, h_\ell, h_r, seq_d, seq_h)$. We have that* $\mathsf{VerifPoP}_{AVL}(h, d, (H, f, h_\ell, h_r, seq_d, seq_h)) = \mathsf{true}$ *if, and only if, there exists a node in T whose data is d and $(H, f, h_\ell, h_r, seq_d, seq_h)$ is the proof of presence of d in T.*

PROOF. We do the proof by induction on the size of seq_d, *i.e.* $|seq_d|$.

Base case $|seq_d| = 0$: In such a case, $seq_d = [\,]$. We prove the two sides of the equivalence.

Assume first that $\mathsf{VerifPoP}_{AVL}(h, d, (H, f, h_\ell, h_r, seq_d, seq_h)) = \mathsf{true}$. Hence, by Definition C.5, it means that $|seq_h| = |seq_d|$ and $h = \mathsf{CompPoP}_{AVL}(seq_d, seq_h, \mathsf{h}(d, H, f, h_\ell, h_r), d)$. However, since $seq_d = []$, the definition also indicates us that $\mathsf{CompPoP}_{AVL}([\,], [\,], \mathsf{h}(d, H, f, , h_\ell, h_r), d) = \mathsf{h}(d, H, f, , h_\ell, h_r)$. Hence $h = \mathsf{CompPoP}_{AVL}(seq_d, seq_h, \mathsf{h}(d, H, f, h_\ell, h_r), d)$ implies that $\mathsf{h}(d, H, f, h_\ell, h_r) = h$. But we know that T is an AVL hash tree whose hash value is h, hence it implies that the hash value of the root of T is $h = \mathsf{h}(d, H, f, h_\ell, h_r)$. Therefore, there exists a node in T whose data is d. Moreover, by Definition C.4, since the root has d as stored value, it implies that $(H, f, h_\ell, h_r, seq_d, seq_h)$ is the proof of presence of d in T.

Let's now assume that there exists a node in T whose stored data is d and $(H, f, h_\ell, h_r, seq_d, seq_h)$ is the proof of presence of d in T. Let's denote by n_0 the node in T whose data is d, *i.e.* n_0 is labeled by $(d, \mathsf{h}(d, H, f, h_\ell, h_r))$. Since $(H, f, h_\ell, h_r, seq_d, seq_h)$ is the proof of presence of d in T and $|seq_d| = 0$, then we deduce by Definition C.4 that n_0 is the root of T, $seq_d = [\ ]$ and $seq_h = [\ ]$. In such a case, $\mathsf{CompPoP}_{AVL}(seq_d, seq_h, \mathsf{h}(d, H, f, h_\ell, h_r), d) = \mathsf{CompPoP}_{AVL}([\,], [\,], \mathsf{h}(d, H, f, h_\ell, h_r), d) = \mathsf{h}(d, H, f, h_\ell, h_r)$. But n_0 being the root of T indicates that $h = \mathsf{h}(d, H, f, h_\ell, h_r)$

and so we conclude that $\mathsf{CompPoP}_{AVL}(seq_d, seq_h, \mathsf{h}(d, H, f, h_\ell, h_r), d) = h$. Therefore, $\mathsf{VerifPoP}_{AVL}(h, d, (H, f, h_\ell, h_r, seq_d, seq_h)) = \mathsf{true}$, thus the result holds.

Inductive step $|seq_d| > 0$: Assume first that $\mathsf{VerifPoP}_{AVL}(h, d, (H, f, h_\ell, h_r, seq_d, seq_h)) = \mathsf{true}$. By Definition C.5, it means that $h = \mathsf{CompPoP}_{AVL}(seq_d, seq_h, \mathsf{h}(d, H, f, h_\ell, h_r), d)$. However, $\mathsf{CompPoP}_{AVL}(seq_d, seq_h, \mathsf{h}(d, H, f, h_\ell, h_r), d)$ is a partial function hence we do a case analysis on the possible value of seq_d and seq_h:

— *Case 1: $seq_d = (d_1, H_1, f_1) :: seq_d$, $seq_h = h_1 :: seq'_h$, $|seq'_d| = |seq'_h|$, $d \mathcal{R} d_1$ and* $\mathsf{CompPoP}_{AVL}(seq_d, seq_h, h', d) = \mathsf{CompPoP}_{AVL}(seq'_d, seq'_h, \mathsf{h}(d_1, H_1, f_1, h', h_1), d_1)$ *where* $h' = \mathsf{h}(d, H, f, h_\ell, h_r)$. Since we have that $\mathsf{CompPoP}_{AVL}(seq'_d, seq'_h, \mathsf{h}(d_1, H_1, f_1, h', h_1), d_1) = h$, we deduce that $\mathsf{VerifPoP}_{AVL}(h, d_1, (H_1, f_1, h', h_1, seq'_d, seq'_h)) = \mathsf{true}$. Thus by inductive hypothesis, we deduce that there exists a node n_0 in T whose stored data is d_1 and $(H_1, f_1, h', h_1, seq'_d, seq'_h)$ is the proof of presence of d_1 in T. Hence, by Definition C.4, we deduce that n_0 is labeled by $(d_1, \mathsf{h}(d_1, H_1, f_1, h', h_1))$ where h' (resp. h_1) is the hash value of the left (resp. right) child of n_0 if it exists, else $h' = \mathsf{null}$ (resp. $h_1 = \mathsf{null}$). But $h' = \mathsf{h}(d, H, f, h_\ell, h_r)$. Hence by definition of an AVL hash tree, we deduce that the left child of n_0 exists and that his stored value is d and so there exists a node n_1 in T whose data is d.
It remains to prove that $(H, f, h_\ell, h_r, seq_d, seq_h)$ is the proof of presence of d in T. We already know that $(H_1, f_1, h', h_1, seq'_d, seq'_h)$ is the proof of presence of d_1 in T where $h' = \mathsf{h}(d, H, f, h_\ell, h_r)$. But d_1 is the data stored by n_0 and n_1 is the left child of n_0 whose data stored is d. Thus the path to n_1 contains the path to n_0. Moreover, we know that h_1 is the hash value of the right child of n_0 if it exists or else $h_1 = \mathsf{null}$. Hence h_1 is the hash value of the sibling of n_1 if it exists or else $h_1 = \mathsf{null}$. Therefore, we can deduce by Definition C.4 that $(H, f, h_\ell, h_r, seq_d, seq_h)$ is a proof of presence of d in T.

— *Case 2: $seq_d = (d_1, H_1, f_1) :: seq_d$, $seq_h = h_1 :: seq'_h$, $|seq'_d| = |seq'_h|$, $d_1 \mathcal{R} d$ and* $\mathsf{CompPoP}_{AVL}(seq_d, seq_h, h', d) = \mathsf{CompPoP}_{AVL}(seq'_d, seq'_h, \mathsf{h}(d_1, H_1, f_1, h_1, h'), d_1)$ *where* $h' = \mathsf{h}(d, H, f, h_\ell, h_r)$. Similar to Case 1.

We now to prove the other implication of the equivalence. Assume that there exists a node in T whose data is d and $(H, f, h_\ell, h_r, seq_d, seq_h)$ is the proof of presence of d in T. By Definition C.4, we know that $|seq_h| = |seq_d|$ thus let us denote $seq_h = h_1 :: seq'_h$ and $seq_d = (d_1, H_1, f_1) :: seq'_d$. Moreover, it implies that there exist a node n_0 in T labeled by $(d, \mathsf{h}(d, H, f, h_\ell, h_r))$ and if $n_k, \dots, n_0$ is the path to n_0 in T then for all $i \in \{1, \dots, k\}$,

— n_i is labeled by $(d_i, \mathsf{h}(d_i, H_i, f_i, h', h''))$ for some $h', h'' \in \mathcal{H}$; and
— h_i is the hash value of $\mathsf{Sib}_T(n_{i-1})$ if it exists, else $h_i = \mathsf{null}$.

We have to verify that $\mathsf{VerifPoP}_{AVL}(h, d, (H, f, h_\ell, h_r, seq_d, seq_h)) = \mathsf{true}$, *i.e.* we have to show that $h = \mathsf{CompPoP}_{AVL}(seq_d, seq_h, \mathsf{h}(d, H, f, h_\ell, h_r), d)$. Since we assumed $|seq_d| > 0$, we can deduce that n_0 is not the root of T. We do a case analysis on whether $d \mathcal{R} d_1$ or $d_1 \mathcal{R} d$.

— *Case $d \mathcal{R} d_1$:* In such a case, let us consider the label of n_1. We know that d_1 is the data stored in n_1. But we assumed that $d \mathcal{R} d_1$, then by definition of an AVL hash tree, it implies that n_0 is the left child of n_1 and so n_1 is labeled by $(d_1, \mathsf{h}(d_1, H_1, f_1, h', h_1))$ where $h' = \mathsf{h}(d, H, f, h_\ell, h_r)$. Therefore, by Definition C.4, we deduce that $(H_1, f_1, h', h_1, seq'_d, seq'_h)$ is the proof of presence of d_1 in T. By our inductive hypothesis, it implies that $\mathsf{VerifPoP}_{AVL}(h, d_1, (H_1 f_1 h', h_1, seq'_d, seq'_h,), =)\mathsf{true}$ and so $\mathsf{CompPoP}_{AVL}(seq'_d, seq'_h, \mathsf{h}(d_1, H_1, f_1, h', h_1)), d_1) = h$. But by Definition C.5, we conclude that $\mathsf{CompPoP}_{AVL}(seq_d, seq_h, h', d) = h$ and so the result holds.
— *Case $d_1 \mathcal{R} d$:* Similar to the previous case. □

C.3. Proof of absence

Thanks to the combination of the binary search tree and Merkle tree, we can prove that some data is not in the data structure, called proof of absence. In this section, given a relation $\mathcal{R}$ over a set D, we say that $d, d' \in D$ are equal w.r.t. $\mathcal{R}$ if $d \mathcal{R} d'$ and $d' \mathcal{R} d$. Moreover, we say that d is related to d' w.r.t. $\mathcal{R}$ if $d \mathcal{R} d'$ or $d' \mathcal{R} d$. Lastly, we say that $\mathcal{R}$ is a strict order over D if for all $d, d' \in D$, d is related but not equal to d' w.r.t. $\mathcal{R}$.

Definition C.6 (*Proof of absence*). Let D be a set partially ordered by a relation $\mathcal{R}$. Let $d \in D$. Let T be an AVL hash tree such that $\mathcal{R}$ is a strict order over the data stored in T and such that d is related but not equal w.r.t. $\mathcal{R}$ to the stored data of all nodes in T. Let $n_k, \dots, n_1$ be the path of the node n_1 in T such that for all $i \in \{2, \dots, k\}$, if d_i is the label of n_i and n_{i-1} is the left child (resp. right child) of n_i then $d \mathcal{R} d_i$ (resp. $d_i \mathcal{R} d$). Moreover, if $d \mathcal{R} d_1$ (resp. $d_1 \mathcal{R} d$) then n_1 does not have a left (resp. right) child. The *proof of absence of* d *in* T is defined as the tuple $(h_\ell, h_r, (d_1, H_1, f_1) :: seq_d, seq_h)$ such that $(H_1, f_1, h_\ell, h_r, seq_d, seq_h)$ is the proof of presence of d_1 in T.

The main idea of the proof of absence is to provide the path where a node labeled by d should be in T if d were in the tree. As such, since a data can always be inserted below a node, the path is necessary from the root to a node that has at most one child node. Moreover, since the data in the tree are ordered thanks to the property of the AVL tree, this path guarantees the absence of the data in the tree.

Example C.5. Consider the AVL tree T from Example C.3. Consider some data d is not equal but related w.r.t. $\mathcal{R}$ to d_i, for all $i \in \{1, \dots, 6\}$. Moreover, consider that $d_5 \mathcal{R} d \mathcal{R} d_6$. The proof of absence of d in T is the tuple $(h_4, \mathsf{null}, seq_d, seq_h)$ where:

— $seq_d = [(d_5, 1, 1), (d_6, 2, 1), (d_3, 3, -1)]$
— $seq_h = [\mathsf{h}(d_7, \mathsf{null}, \mathsf{null}), \mathsf{h}(d_1, 1, -1, \mathsf{null}, \mathsf{h}(d_2, \mathsf{null}, \mathsf{null}))]$

Note that the proof of absence $(h_4, \mathsf{null}, seq_d, seq_h)$ is in fact the proof of presence $(1, 1, h_4, \mathsf{null}, seq'_d, seq_h)$ of d_5 in T where $seq'_d = [(d_6, 2, 1), (d_3, 3, -1)]$. $\diamond$

The verification of a proof of absence is very similar to the verification of a proof of presence as expected.

Definition C.7 (*Verification of a proof of absence*). Let D be a set partially ordered by a relation $\mathcal{R}$. Let $d \in D$, $h \in \mathcal{H}$ and a tuple $(h_\ell, h_r, seq_d, seq_h)$. The verification that $(h_\ell, h_r, seq_d, seq_h)$ proves the absence of d in h, denoted $\mathsf{VerifPoAbs}_{AVL}(h, d, (h_\ell, h_r, seq_d, seq_h))$, is defined as follows:

We have that $\mathsf{VerifPoAbs}_{AVL}(h, d, (h_\ell, h_r, seq_d, seq_h)) = \mathsf{true}$ if, and only if, $|seq_d| > 0$, $seq_d = [(d_1, H_1, f_1), \dots, (d_n, H_n, f_n)]$, $\mathsf{VerifPoP}_{AVL}(h, d_1, (H_1, f_1, h_\ell, h_r, seq'_d, seq_h)) = \mathsf{true}$ where $seq'_d = [(d_2, H_2, f_2), \dots, (d_n, H_n, f_n)]$ and for all $i \in \{2, n\}$,

— $d_{i-1} \mathcal{R} d_i$ implies that $d \mathcal{R} d_i$ and $d_i \not\mathcal{R} d$
— $d_i \mathcal{R} d_{i-1}$ implies that $d_i \mathcal{R} d$ and $d \not\mathcal{R} d_i$
— $d \mathcal{R} d_1$ or exclusive $d_1 \mathcal{R} d$
— $d \mathcal{R} d_1$ (resp. $d_1 \mathcal{R} d$) implies that $h_\ell = \mathsf{null}$ (resp. $h_r = \mathsf{null}$)

The verification of a proof of absence is composed of two parts. The first one consist of verifying that the node labeled by d_1 is indeed a leaf in the tree whereas the second part consists of verifying that that the bitstring d follows the path of the leaf labeled by d_1. By verifying that d is not the stored data of any of the node in the path, we ensure that d is absent from the AVL hash tree represented by h.

Example C.6. Coming back to Example C.5, we have that $\mathsf{VerifPoAbs}_{AVL}(h, d, (h_4, \mathsf{null}, seq_d, seq_h)) = \mathsf{true}$ proves the absence of d in the Tree represented by h. Indeed, $\mathsf{VerifPoP}_{AVL}(h, d_5, (1, 1, h_4, \mathsf{null}, seq'_d, seq_h)) = \mathsf{true}$. Moreover, we know that $d_5 \mathcal{R} d \mathcal{R} d_6$ with d not equal w.r.t. $\mathcal{R}$ to d_5 or d_6 therefore we have

— $d_5 \mathcal{R} d$ and $d \not\mathcal{R} d_5$; moreover the second element of $(h_4, \mathsf{null}, seq_d, seq_h)$ is the constant null.
— $d \mathcal{R} d_6$ and $d_6 \not\mathcal{R} d$
— $d_3 \mathcal{R} d$ and $d \not\mathcal{R} d_3$ $\quad\diamond$

We can now prove the soundness and completeness of the verification of a proof of absence as stated in the following lemma.

LEMMA C.4. *Let D be a set partially ordered by a relation $\mathcal{R}$. Let T be an AVL hash tree over D whose hash value is h and such that $\mathcal{R}$ is a strict order over the data stored in T. Let $d \in D$ and $(h_\ell, h_r, seq_d, seq_h)$ be a tuple. We have that $\mathsf{VerifPoAbs}_{AVL}(h, d, (h_\ell, h_r, seq_d, seq_h)) = \mathsf{true}$ if, and only if, d is related but not equal w.r.t. $\mathcal{R}$ to the data of any node in T and $(h_\ell, h_r, seq_d, seq_h)$ is the proof of absence of d in T.*

PROOF. Consider first that $\mathsf{VerifPoAbs}_{AVL}(h, d, (h_\ell, h_r, seq_d, seq_h)) = \mathsf{true}$. By Definition C.7, it means that $|seq_d| > 0$, $seq_d = [(d_1, H_1, f_1), \dots, (d_k, H_k, f_k)]$, $\mathsf{VerifPoP}_{AVL}(h, d_1, (H_1, f_1, h_\ell, h_r, seq'_d, seq_h)) = \mathsf{true}$ where $seq'_d = [(d_2, H_2, f_2), \dots, (d_k, H_k, f_k)]$ and for all $i \in \{2, k\}$,

— $d_{i-1} \mathcal{R} d_i$ implies that $d \mathcal{R} d_i$ and $d_i \not\mathcal{R} d$
— $d_i \mathcal{R} d_{i-1}$ implies that $d_i \mathcal{R} d$ and $d \not\mathcal{R} d_i$
— $d \mathcal{R} d_1$ or exclusive $d_1 \mathcal{R} d$
— $d \mathcal{R} d_1$ (resp. $d_1 \mathcal{R} d$) implies that $h_\ell = \mathsf{null}$ (resp. $h_r = \mathsf{null}$)

By Lemma C.3, we deduce that that $(H_1, f_1, h_\ell, h_r, seq'_d, seq_h)$ is the proof of presence of d_1 in T and so by Definition C.4, if $n_k, \dots, n_1$ is the path to n_1 then for all $i \in \{1, \dots, k\}$, n_i is labeled by $(d_i, \mathsf{h}(d_i, H_i, f_i, h, h'))$ for some $h, h' \in \mathcal{H}$.

Let us now show that d is related but not equal w.r.t. $\mathcal{R}$ to the stored data of any node in T. By hypothesis, we know that $\mathcal{R}$ is a strict oder over thel data stored in T. Let us do a case analysis:

Let d' be the data stored in a node of T. If $d' = d_i$ for some $i \in \{1, \dots, k\}$ then the result trivially holds since we have that d is related but not equal to d_i.

In such a case, we deduce that $d' \mathcal{R} d_1$ or exclusive $d_1 \mathcal{R} d'$. Let's

Moreover, we trivially have by Definition C.6 that $(seq_d, seq_h$,,i)s the proof of absence of d in the T. In fact since Lemma C.3 is an equivalence, we directly have that if $(seq_d, seq_h$,,i)s

$\square$

C.3.1. Proof of presence and absence. Since an AVL LexTree is a LexTree, so the proof of presence and absence is very similar as what we defined in the previous section – the only difference is that to be able to compute the hash value of the root, we need to add the additional data (H, f) in the proof. In addition, the verification process is more complex since AVL LexTree is a concrete example and thus has more specific properties (e.g. the correctness of height and height factor in a node) other than the generic property.

Definition C.8. Let $seq = [(d_1, H_1, f_1); \dots; (d_n, H_n, f_n)]$ be a sequence of bitstrings. The verification that seq corresponds to the data of a path (from a node to the root) in an AVL LexTree, denoted $\mathsf{VerifData}_{AVL}(seq)$, is defined as follows:

We have that $\mathsf{VerifData}_{AVL}(seq) = \mathsf{true}$ if, and only if, $H_1 = 0$ implies $f_1 = 0$; and for all $i \in \{2, \dots, n\}$, if $d_{i-1} \mathcal{R} d_i$ then $H_i = H_{i-1} + \max(1, 1 - f_i)$ else $H_i = H_{i-1} + \max(1, 1 + f_i)$.

Intuitively, $\mathsf{VerifData}_{AVL}(seq)$ is the verification that whether the elements in the given sequence seq are corresponding to a valid path of an AVL LexTree. This will be useful for verifying the proof of presence, proof of absence, and the random checking of the existance of an AVL LexTree.

Example C.7. Let $seq = [(d_5, 0, 0), (d_4, 1, 0), (d_2, 2, -1)]$ such that $d_2 \mathcal{R} d_4 \mathcal{R} d_5$. If we denote this sequence by $[\text{data}_i, H_i, f_i]_{i=\{1,2,3\}}$, then we have $H_1 = 0$ and $f_1 = 0$, and $H_2 = H_1 + \max(1, 1 + f_2) = 1$ and $H_3 = H_2 + \max(1, 1 + f_3) = 2$. So, we have that $\mathsf{VerifData}_{AVL}(seq) = \mathsf{true}$. The Figure **??** shows a possible AVL LexTree T' in which the data in seq are the data stored in the nodes of a path from the node with position $w = r \cdot r$ to the root. $\diamond$

Definition C.9. Given two bitstrings d, h and a tuple $pr = (h_\ell, h_r, H, f, seq_d, seq_h)$, the verification that pr proves the presence of d in h for an AVL LexTree, denoted $\mathsf{VerifPoP}_{AVL}(h, d, pr)$, is defined as follows:
We have that $\mathsf{VerifPoP}_{AVL}(h, d, pr) = \mathsf{true}$ if, and only if, $\mathsf{VerifPoP}_L(h, (d, H, f), (h_\ell, h_r, seq_d, seq_h)) = \mathsf{true}$, $\mathsf{VerifData}_{AVL}((d, H, f) :: seq_d) = \mathsf{true}$, and if we denote seq_d and seq_h respectively by $[(d_i, H_i, f_i)]_{i=1,\dots,k}$ and $[h_i]_{i=1,\dots,k}$ then

— $H = 0$ if, and only if $h_\ell = h_r = \mathsf{null}$; and
— for all $i \in \{2, \dots, k\}$, $h_i \neq \mathsf{null}$; and
— $h_1 = \mathsf{null}$ implies $H = 0$; and
— if either $h_\ell = \mathsf{null}$ or $h_r = \mathsf{null}$ then $H = 1$.

Example C.8.
Come back to Figure **??**, given (d_5, h) and $pr = (h_\ell, h_r, H, f, seq_d, seq_h)$ such that $h = \mathsf{h}((d_2, 2, -1), \mathsf{h}_1, \mathsf{h}((d_4, 1, 0), \mathsf{h}_3, \mathsf{h}_5)))$ and $pr = (\mathsf{null}, \mathsf{null}, 0, 0, [(d_4, 1, 0), (d_2, 2, -1)], [\mathsf{h}_3, \mathsf{h}_1])$, we have that $\mathsf{VerifPoP}_L(h, (d_5, 0, 0), (\mathsf{null}, \mathsf{null}, [(d_4, 1, 0), (d_2, 2, -1)], [\mathsf{h}_3, \mathsf{h}_1])) = \mathsf{true}$ by applying the verification process defined in Def. C.5. In addition, we have that $\mathsf{VerifData}_{AVL}((d_5, 0, 0), (d_4, 1, 0), (d_2, 2, -1)) = \mathsf{true}$ showed by the example C.7. Moreover, we denote $[\mathsf{h}_3, \mathsf{h}_1]$ by $[\text{hash}_1, \text{hash}_2]$, then we have

— $H = 0$ and $h_\ell = h_r = \mathsf{null}$;
— $\text{hash}_2 \neq \mathsf{null}$.

So, we have $\mathsf{VerifPoP}_{AVL}(h, d, pr) = \mathsf{true}$. $\diamond$

Definition C.10. Given two bitstrings d, h and a tuple $(h_\ell, h_r, seq_d, seq_h)$. The verification that $(h_\ell, h_r, seq_d, seq_h)$ proves the absence of d in h for an AVL LexTree, denoted $\mathsf{VerifPoAbs}_{AVL}(h, d, (h_\ell, h_r, seq_d, seq_h))$, is defined as follows:
We have that $\mathsf{VerifPoAbs}_{AVL}(h, d, (h_\ell, h_r, seq_d, seq_h)) = \mathsf{true}$ if, and only if, $\mathsf{VerifPoAbs}_L(h, d, (h_\ell, h_r, seq_d, seq_h)) = \mathsf{true}$, $\mathsf{VerifData}_{AVL}(seq_d) = \mathsf{true}$, and if we denote seq_d and seq_h respectively by $[(d_i, H_i, f_i)]_{i=1,\dots,k}$ and $[h_i]_{i=1,\dots,k'}$ then $k = k' + 1$, for all $i \in \{2, \dots, k'\}$, $h_i \neq \mathsf{null}$, $H_1 = |f_1|$ and $f_1 = -1$ (resp. 0 and 1) implies $h_\ell = \mathsf{null}, h_r \neq \mathsf{null}$ (resp. $h_\ell = \mathsf{null}, h_r = \mathsf{null}$ and $h_\ell \neq \mathsf{null}, h_r = \mathsf{null}$).

Example C.9. Come back to Figure **??**, given (d, h) and $pr = (h_\ell, h_r, H, f, seq_d, seq_h)$ such that $d_4 \mathcal{R} d_5 \mathcal{R} d$, $h = \mathsf{h}((d_2, 2, -1), \mathsf{h}_1, \mathsf{h}((d_4, 1, 0), \mathsf{h}_3, \mathsf{h}_5)))$ and $pr = (\mathsf{null}, \mathsf{null}, [(d_5, 0, 0), (d_4, 1, 0), (d_2, 2, -1)], [\mathsf{h}_3, \mathsf{h}_1])$, we have that $\mathsf{VerifPoP}_L(h, (d_5, 0, 0), (\mathsf{null}, \mathsf{null}, [(d_4, 1, 0), (d_2, 2, -1)], [\mathsf{h}_3, \mathsf{h}_1])) = \mathsf{true}$ as stated in the previous example, and we have

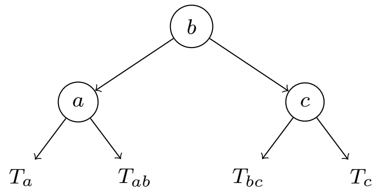
— $d_4 \mathcal{R} d_5$, $d_2 \mathcal{R} d_4$, and $d_2 \mathcal{R} d$, $d_4 \mathcal{R} d$, and $d \not\mathcal{R} d_2$ and $d \not\mathcal{R} d_4$;
— $d \not\mathcal{R} d_5$
— $d_5 \mathcal{R} d$ and $h_r = \mathsf{null}$

So, $\mathsf{VerifPoAbs}_L(h, d, (h_\ell, h_r, seq_d, seq_h)) = \mathsf{true}$. In addition,we have $\mathsf{VerifData}_{AVL}((d_5, 0, 0), (d_4, 1, 0), (d_2, 2, -1)) = \mathsf{true}$ showed by the example C.7. Moreover, $|seq_d| = 3$ and $|seq_h| = 2$, so $|seq_d| = |seq_h| + 1$. Furthermore, if we denote $[(d_5, 0, 0), (d_4, 1, 0), (d_2, 2, -1)]$ by $[(\mathrm{data}_1, H_1, f_1), (\mathrm{data}_2, H_2, f_2), (\mathrm{data}_3, H_3, f_3)]$, and $[\mathsf{h}_3, \mathsf{h}_1]$ by $[\mathrm{hash}_1, \mathrm{hash}_2]$, then we have $\mathrm{hash}_2 \neq \mathsf{null}$, $H_1 = |f_1| = 0$, and $h_\ell = h_r = \mathsf{null}$. So, $\mathsf{VerifPoAbs}_{AVL}(h, d, (h_\ell, h_r, seq_d, seq_h)) = \mathsf{true}$. $\diamond$

Definition C.11. Given the bitstrings d, d', h, h' and a bitstring pr, the verification that pr proves the modification of d in h into d' in h', denoted $\mathsf{VerifPoM}_{AVL}((h, d), (h', d'), pr)$, is defined as follows:
We have that $\mathsf{VerifPoM}_{AVL}((h, d), (h', d'), pr) = \mathsf{true}$ if, and only if, $\mathsf{VerifPoP}_{AVL}(h, d, pr) = \mathsf{true}$ and $\mathsf{VerifPoP}_{AVL}(h', d', pr) = \mathsf{true}$.

Intuitively, the above definition specifies how to verify the proof that an AVL LexTree T' whose root value is h' is obtained by replacing data d with d' in the AVL LexTree T represented by h.

C.3.2. (Proof of) addition. Let T be an AVL tree over a set $\mathcal{D}$ ordered by $\mathcal{R}$. To add a new label $d \in \mathcal{D}$ (i.e., d is not a label of a node in T), we follow the classical procedure that is described below. Let w be the unique position such that the tree $T' = T \cup \{(d, w)\}$ is a binary search tree, and let $n_1, \cdots, n_k$ be the path from the node (d, w) up to the root in T'. If T' is an AVL tree, then the output is $\mathsf{add}(T, d) = T'$. Otherwise, let i be the smallest index such that the subtree with root n_i is not AVL. The function $\mathsf{ord} : \{a, b, c\} \to \{n_{i-2}, n_{i-1}, n_i\}$ maps a name to the nodes n_i, n_{i-1} and n_{i-2} such that $\mathsf{lbl}(a) \,\mathcal{R}\, \mathsf{lbl}(b) \,\mathcal{R}\, \mathsf{lbl}(c)$. To compute $\mathsf{add}(T, d)$, we consider four subtrees of T': (1) subtree T_a starting in the left child of the node a. Note that $x \,\mathcal{R}\, a$ for all labels x in T_a. (2) subtree T_{ab} starting either in the right child of a or in the left child of b such that $a \,\mathcal{R}\, x \,\mathcal{R}\, b$ for all labels x in T_{ab}. (3) subtree T_{bc} starting either in the right child of b or in the left child of c such that $b \,\mathcal{R}\, x \,\mathcal{R}\, c$ for all labels x in T_{bc}. (4) subtree T_c starting in the right child of the node c where $c \,\mathcal{R}\, x$ for all labels x in T_c. The tree $\mathsf{add}(T, d)$ is constructed from T' by replacing the subtree rooted by n_i is with the following subtree:

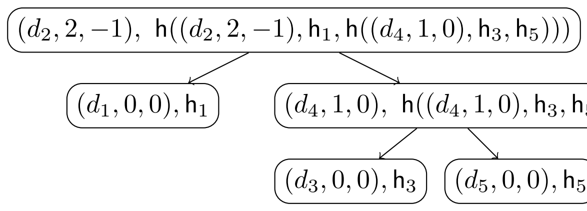


Let T be an AVL LexTree over a set $\mathcal{D}$ ordered by $\mathcal{R}$. Adding a new data $d \in \mathcal{D}$ (i.e., d is not a data of a node in T) in T, denoted $\mathsf{add}_{Lex}(T, d)$, corresponds to outputting the following AVL LexTree:

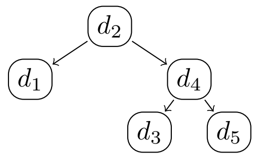
$$\mathsf{add}_{Lex}(T, d) = \mathsf{to_h}(\mathsf{add}(\mathsf{to_h}^{-1}(T), d))$$

Example C.10. Figure 10 presents an example to add d_6 into AVL LexTree T_{AVL}. Basically, AVL tree T can be obtained by applying $\mathsf{to_h}^{-1}(T_{AVL})$, and T' is the tree after adding d_6 into T, but T' is not an AVL tree. By balancing T', we obtain AVL tree T''. Then, we can convert AVL tree T'' to AVL LexTree T'_{AVL}. $\diamond$

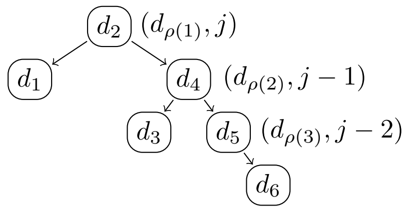
Definition C.12. Let T and T' be two AVL trees over a set $\mathcal{D}$ ordered by $\mathcal{R}$. Let $d \in \mathcal{D}$ be such that d is not a label of a node in T. The proof that adding d in T outputs T' is the proof of absence of d in T.


$(d_2, 2, -1),\ \mathsf{h}((d_2, 2, -1), \mathsf{h}_1, \mathsf{h}((d_4, 1, 0), \mathsf{h}_3, \mathsf{h}_5)))$
$(d_1, 0, 0), \mathsf{h}_1$
$(d_4, 1, 0),\ \mathsf{h}((d_4, 1, 0), \mathsf{h}_3, \mathsf{h}_5)$
$(d_3, 0, 0), \mathsf{h}_3$
$(d_5, 0, 0), \mathsf{h}_5$

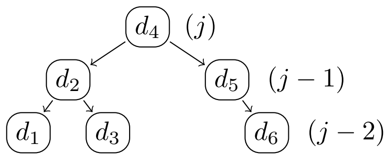
(a) AVL LexTree T_{AVL}



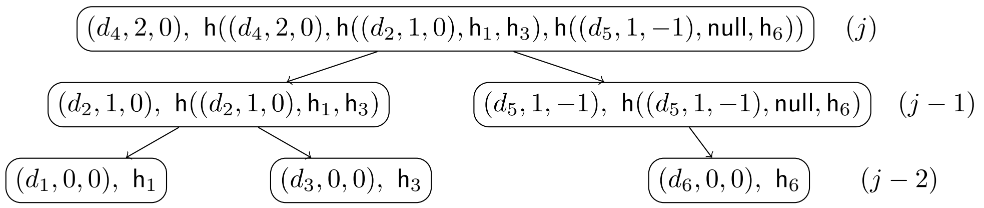
(b) AVL tree T



(c) tree T'



(d) AVL tree T''


$(d_4, 2, 0),\ \mathsf{h}((d_4, 2, 0), \mathsf{h}((d_2, 1, 0), \mathsf{h}_1, \mathsf{h}_3), \mathsf{h}((d_5, 1, -1), \mathsf{null}, \mathsf{h}_6))$ (j)
$(d_2, 1, 0),\ \mathsf{h}((d_2, 1, 0), \mathsf{h}_1, \mathsf{h}_3)$
$(d_5, 1, -1),\ \mathsf{h}((d_5, 1, -1), \mathsf{null}, \mathsf{h}_6)$ $(j-1)$
$(d_1, 0, 0),\ \mathsf{h}_1$
$(d_3, 0, 0),\ \mathsf{h}_3$
$(d_6, 0, 0),\ \mathsf{h}_6$ $(j-2)$

(e) AVL tree T'_{AVL}

Fig. 10. Examples of adding data d_6 into an AVL LexTree T_{AVL}, and obtains T'_{AVL}. In this figure, AVL tree T is converted from T_{AVL} i.e. $T = \mathsf{to}_{\mathsf{h}}^{-1}(T_{AVL})$, T' is the tree after inserting d_6 but before balancing the tree. T'' is an AVL tree obtained by balancing T', and T'_{AVL} is the AVL LexTree converted from T'' i.e. $T'_{AVL} = \mathsf{to}_{\mathsf{h}}(T'')$. In addition, we have that $\mathsf{h}_i = \mathsf{h}((d_i, 0, 0), \mathsf{null}, \mathsf{null})$ for all $i = \{1, 3, 5, 6\}$. Moreover, for all $i \in [1, 5]$, we have $d_i \mathcal{R} d_{i+1}$ and $d_{i+1} \not\mathcal{R} d_i$.

Example C.11. Come back to Fig. 10, according to the Def. C.12, we have that the proof of adding d_6 into T_{AVL} outputing T'_{AVL} is the proof of absence of d_6 in T_{AVL}. So, the proof is $(h_\ell, h_r, seq_d, seq_h)$, where $h_\ell = h_r = \mathsf{null}$, and $seq_d = [(d_5, 0, 0), (d_4, 1, 0), (d_2, 2, -1)]$ and $seq_h = [\mathsf{h}_3, \mathsf{h}_1]$. $\diamond$

Definition C.13. Let $(h_\ell, h_r, seq_d, seq_h)$ be a tuple of bitstrings. Let h, h' and d be bitstrings. The verification that (seq_d, seq_h) proves h' to be the result of the addition of d in h, denoted $\mathsf{VerifPoAdd}_{AVL}(h, h', d, (h_\ell, h_r, seq_d, seq_h))$, is defined as follows:

First of all, we will require that $\mathsf{VerifPoAbs}_{AVL}(h, d, (h_\ell, h_r, seq_d, seq_h)) = \mathsf{true}$, that is we verify the absence of d in h. In such a case, we can denote $seq_d = [(d_i, H_i, f_i)]_{i=1,\ldots,k}$ and $seq_h = [h_i]_{i=1,\ldots,k-1}$. Moreover, let's consider that $H_0 = -1$. We now define recursively two functions $\mathsf{CompH}(i)$ and $\mathsf{Compf}(i)$ as follows: $\mathsf{CompH}(0) = 0$, $\mathsf{Compf}(0) = 0$ and for all

$i > 0$, if $d \mathcal{R} d_i$ (resp. $d_i \mathcal{R} d$) then $\mathsf{Compf}(i) = \mathsf{CompH}(i-1) - H_{i-1} + f_i$ (resp. $\mathsf{Compf}(i) = H_{i-1} - \mathsf{CompH}(i-1) + f_i$) and $\mathsf{CompH}(i) = H_i + \max(0; |\mathsf{Compf}(i)| - |f_i|)$.

Let us consider now j the smallest index such that $\mathsf{Compf}(j) \in \{-2, 2\}$. If such j does not exists, *i.e.* it means that the new tree is already AVL, then:

$$\begin{aligned}\mathsf{VerifPoAdd}_{AVL}(h, h', d, (h_\ell, h_r, seq_d, seq_h)) = \ & \mathsf{VerifPoAbs}_{AVL}(h, d, (h_\ell, h_r, seq_d, seq_h)) \wedge \\ & \mathsf{VerifPoP}_{AVL}(h', d, (\mathsf{null}, \mathsf{null}, 0, 0, seq'_d, seq'_h))\end{aligned}$$

where $seq'_d = [(d_i, \mathsf{CompH}(i), \mathsf{Compf}(i))]_{i=1,\ldots,k}$ and $seq'_h = h_\ell :: seq_h$ (resp. $h_r :: seq_h$ and $\mathsf{null} :: seq_h$) if $f_1 = 1$ (resp. -1 and 0).

Otherwise, if j exists then let ρ be a mapping of $\{1, 2, 3\}$ to $\{j, j-1, j-2\}$ such that $d_{\rho(1)} \mathcal{R} d_{\rho(2)} \mathcal{R} d_{\rho(3)}$. We do a case analysis on j:

Case $j = 2$: In such a case, we define $h'_1 = \mathsf{h}((d_{\rho(3)}, 0, 0), \mathsf{null}, \mathsf{null})$ and we have that:

$$\begin{aligned}\mathsf{VerifPoAdd}_{AVL}(h, h', d, (h_\ell, h_r, seq_d, seq_h)) = \ & \mathsf{VerifPoAbs}_{AVL}(h, d, (h_\ell, h_r, seq_d, seq_h)) \wedge \\ & \mathsf{VerifPoP}_{AVL}(h', d_{\rho(1)}, (\mathsf{null}, \mathsf{null}, 0, 0, seq'_d, seq'_h))\end{aligned}$$

where $seq'_d = [(d_{\rho(2)}, 1, 0)]@[(d_i, H_i, f_i)]_{i=3,\ldots,k}$; and $seq'_h = [h'_1]@[h_i]_{i=2,\ldots,k-1}$.

Case $j > 2$: Otherwise, let us consider $h_0 = \mathsf{null}$ and the two hash values h'_{j-1}, h'_{j-2}, the integer f'_{j-1} and the data d'_{j-1} such that: (1) If $\rho(1) = j-2$ (resp. $\rho(3) \neq j-2 \wedge d \mathcal{R} d_{j-2}$) then $h'_{j-1} = \mathsf{h}((d_{\rho(3)}, H_{j-1}, f), h_{\rho(2)-1}, h_{\rho(3)-1})$, $h'_{j-2} = h_{\rho(1)-1}$, $d'_{j-1} = d_{\rho(1)}$, $f'_{j-1} = \mathsf{Compf}(j-2)$ (resp. 0) with $f = 0$ (resp. -1). (2) If $\rho(3) = j-2$ (resp. $\rho(1) \neq j-2 \wedge d_{j-2} \mathcal{R} d$) then $h'_{j-1} = \mathsf{h}((d_{\rho(1)}, H_{j-1}, f), h_{\rho(1)-1}, h_{\rho(2)-1})$, $h'_{j-2} = h_{\rho(3)-1}$, $d'_{j-1} = d_{\rho(3)}$, $f'_{j-1} = \mathsf{Compf}(j-2)$ (resp. 0) with $f = 0$ (resp. 1).

In such a case, we have that:

$$\begin{aligned}\mathsf{VerifPoAdd}_{AVL}(h, h', d, (h_\ell, h_r, seq_d, seq_h)) = \ & \mathsf{VerifPoAbs}_{AVL}(h, d, (h_\ell, h_r, seq_d, seq_h)) \wedge \\ & \mathsf{VerifPoP}_{AVL}(h', d, (\mathsf{null}, \mathsf{null}, 0, 0, seq'_d, seq'_h))\end{aligned}$$

where $seq'_d = [(d_i, \mathsf{CompH}(i), \mathsf{Compf}(i))]_{i=1,\ldots,j-3}@[(d'_{j-1}, H_{j-1}, f'_{j-1}); (d_{\rho(2)}, H_j, 0)]@[(d_i, H_i, f_i)]_{i=j+1,\ldots,k}$; and $seq'_h = [h_i]_{i=0,\ldots,j-4}@[h'_{j-2}; h'_{j-1}]@[h_i]_{i=j,\ldots,k-1}$.

Intuitively, in the above definition if there is an AVL LexTree T such that the data in the given sequence $seq_d = [(d_i, H_i, f_i)]_{i=1,\ldots,k}$ are stored in the node n_i such that all n_i are on the path of n_1 in T, then H_i is the height of the subtree rooted from n_i, and f_i is the corresponding height factor. In addition, $\mathsf{CompH}(i)$ is the height of n_i after inserting a new data but before balancing (if needed) the tree, and $\mathsf{Compf}(i)$ is the corresponding height factor. For example, in the Fig. 10, we have that H and f of the node storing d_4 are respectively 1 and 0 (as showed in Fig. 10(b)), after inserting a new data d_6 (as showed in Fig. 10(c), the T' is a tree after inserting d_6 but has not been balanced yet), $\mathsf{CompH}(\cdot)$ and $\mathsf{Compf}(\cdot)$ of the node storing d_4 are respectively 2 and -1.

Example C.12. Come back to the Fig. 10, given the proof $(h_\ell, h_r, seq_d, seq_h)$ of adding d_6 into T_{AVL} labelled by h outputing T'_{AVL} labelled by h' (presented in Example C.11), where $h_\ell = h_r = \mathsf{null}$, $seq_d = [(d_5, 0, 0), (d_4, 1, 0), (d_2, 2, -1)]$, $seq_h = [\mathsf{h}_3, \mathsf{h}_1]$, we have that $H_0 = -1$, $\mathsf{CompH}(0) = 0$, and $\mathsf{Compf}(0) = 0$. In addition, since $d_i \mathcal{R} d_6$ for all $i \in [1, 5]$, if denote seq_d by $[\mathrm{data}_i, H_i, f_i]_{i=\{1,2,3\}}$ and denote seq_h by $[\mathrm{hash}_1, \mathrm{hash}_2]$, then we have $\mathrm{data}_1 \mathcal{R} \mathrm{data}_2 \mathcal{R} \mathrm{data}_5 \mathcal{R} d_6$. So, $\mathsf{Compf}(i) = H_{i-1} - \mathsf{CompH}(i-1) + f_i$ and $\mathsf{CompH}(i) = H_i + \max(0; |\mathsf{Compf}(i)| - |f_i|)$ for all $i \in [1, 3]$. Thus, we have $\mathsf{Compf}(1) = -1$ and $\mathsf{CompH}(1) = 1$, $\mathsf{Compf}(2) = 1$ and $\mathsf{CompH}(2) = 2$, and $\mathsf{Compf}(3) = -2$ and $\mathsf{CompH}(3) = 3$. So, there when $j = 3$ we have $\mathsf{Compf}(3) = -2$.

Let ρ be a mapping of $\{1, 2, 3\}$ to $\{j, j-1, j-2\}$ such that $d_{\rho(1)} \mathcal{R} d_{\rho(2)} \mathcal{R} d_{\rho(3)}$. We have $d_{\rho(1)} = d_4$ (w.r.t.the case of j), $d_{\rho(2)} = d_5$ (w.r.t. the case of $j-1$), and $d_{\rho(3)} = d_6$ (w.r.t.

the case of $j-2$) as showed in Fig. 10(b). Since $\rho(3)$ regards to $j-2$, it satisfies point (2) in the case $j>2$ in the Def. C.13. So, we have $h'_{j-1} = \mathsf{h}((d_2,1,0),\mathsf{h}_{h_1},\mathsf{h}_{h_3})$, $h'_{j-2} = \mathsf{null}$, $d'_{j-1} = d_5$, and $f'_{j-1} = -1$.

According to Def. C.13, we need to verify that

$$\begin{aligned}\mathsf{VerifPoAdd}_{AVL}(h,h',d,(h_\ell,h_r,seq_d,seq_h)) = \ & \mathsf{VerifPoAbs}_{AVL}(h,d,(h_\ell,h_r,seq_d,seq_h))\wedge\\ & \mathsf{VerifPoP}_{AVL}(h',d,(\mathsf{null},\mathsf{null},0,0,seq'_d,seq'_h))\end{aligned}$$

Where $seq'_d = [(d_5,1,-1),(d_4,2,0)]$ and $seq'_h = [\mathsf{null},\mathsf{h}((d_2,1,0),\mathsf{h}_{h_1},\mathsf{h}_{h_3})]$.

We will first verify whether $\mathsf{VerifPoAbs}_{AVL}(h,d_6,(h_\ell,h_r,seq_d,seq_h))$ is true, where $h = \mathsf{h}((d_2,2,-1),\mathsf{h}_1,\mathsf{h}((d_4,1,0),\mathsf{h}_3,\mathsf{h}_5))$, $h_\ell = h_r = \mathsf{null}$, $seq_d = [(d_5,0,0),(d_4,1,0),(d_2,2,-1)]$, $seq_h = [\mathsf{h}_3,\mathsf{h}_1]$. This is verified in example C.9.

Now we need to verify that whetehr $\mathsf{VerifPoP}_{AVL}(h',d,(\mathsf{null},\mathsf{null},0,0,seq'_d,seq'_h))$ is true.

We have that $|seq'_d| = |seq'_h| = 2$, and

$$\begin{aligned}&\mathsf{CompPoP}_L([(d_5,1,-1),(d_4,2,0)],[\mathsf{null},\mathsf{h}((d_2,1,0),\mathsf{h}_{h_1},\mathsf{h}_{h_3})],\mathsf{h}((d_6,0,0),\mathsf{null},\mathsf{null}),(d_6,0,0))\\ &=\\ &\mathsf{CompPoP}_L([(d_4,2,0)],[\mathsf{h}((d_2,1,0),\mathsf{h}_{h_1},\mathsf{h}_{h_3})],\mathsf{h}((d_5,1,-1),\mathsf{null},\mathsf{h}((d_6,0,0),\mathsf{null},\mathsf{null})),(d_5,1,-1))\\ &=\\ &\mathsf{CompPoP}_L([],[],\mathsf{h}((d_4,2,0),\mathsf{h}((d_2,1,0),\mathsf{h}_1,\mathsf{h}_3),\mathsf{h}((d_5,1,-1),\mathsf{null},\mathsf{h}_6)),(d_4,2,0))\\ &=\text{h'}\end{aligned}$$

So, we have that $\mathsf{VerifPoP}_L(h',(d_6,0,0),\mathsf{null},\mathsf{null},seq'_d,seq'_h) = \mathsf{true}$. In dadition, if we denote $[(d_6,0,0),(d_5,1,-1),(d_4,2,0)]$ by $[\text{data}_i,H_i,f_i]_{i=\{1,2,3\}}$, then we have $H_1 = 0$ and $f_1 = 0$, and $H_2 = H_1 + \max(1,1+f_2) = 1$ and $H_3 = H_2 + \max(1,1+f_3) = 2$. So, we have that $\mathsf{VerifData}_{AVL}((d_6,0,0) :: seq'_d) = \mathsf{true}$. Moreover, if we denote seq'_h by $[\text{hash}_1,\text{hash}_2]$, then we have

— $H_1 = 0$ and the label of left and right child nodes of the node storing $(d_6,0,0)$ are both null;
— $\text{hash}_2 \neq \mathsf{null}$.

So, we have $\mathsf{VerifPoP}_{AVL}(h',d_6,pr) = \mathsf{true}$.

Since we have $\mathsf{VerifPoAbs}_{AVL}(h,(d_6,0,0),(h_\ell,h_r,seq_d,seq_h)) = \mathsf{true}$ and $\mathsf{VerifPoP}_{AVL}(h',(d_6,0,0),(\mathsf{null},\mathsf{null},0,0,seq'_d,seq'_h)) = \mathsf{true}$, so we have $\mathsf{VerifPoAdd}_{AVL}(h,h',(d_6,0,0),(h_\ell,h_r,seq_d,seq_h)) = \mathsf{true}$ $\diamond$

LEMMA C.5. *Let T be an AVL LexTree whose hash value is h. Let d and h' be two bitstring and (h_ℓ,h_r,seq_d,seq_h) be a tuple. We have that* $\mathsf{VerifPoAdd}_{AVL}(h,h',d,(h_\ell,h_r,seq_d,seq_h)) = \mathsf{true}$ *if, and only if, there exists an AVL LexTree T' labeled by h' such that* $\mathsf{add}_{Lex}(T,d) = T'$ *and (h_ℓ,h_r,seq_d,seq_h) is the proof the adding d to T outputs T'.*

C.4. Random Checking

Since we will use two tree structures to organise the log, we will need the proof that these two trees are presenting the same set of data. By the random checking of the exsitance of a ChronTree (presented in the section **??**), we can ensure that given hash value h and integer N, there exists a ChronTree whose hash value is h and whose size is N. To verify that whether a ChronTree and a LexTree represent the same log, we need to verify that they have stored the same set of data. One way to verify it is to re-build trees according to the given sequence of data, however, the time and size for this verification is $O(N)$, which is too expensive for a single user. To make the verification efficient, we will use random checking. We now explain how the random checking on the AVL LexTree could cover all the data.

Definition C.14. We define the function $\mathsf{Rand}\exists_{AVL}$ such that for all bitstrings h and $pr = (h_\ell, h_r, seq_d, seq_h)$, for all $H \in \mathbb{N}$ and for all $w \in \{\ell, r\}^*$, $\mathsf{Rand}\exists_{AVL}(w, H, h, pr) = \mathsf{true}$ if, and only if, we can denote $seq_d = [(d_i, H_i, f_i)]_{i=1,\dots,k}$ and $w = a_1 \cdot \ldots \cdot a_{k'}$ such that

— $\mathsf{VerifPoP}_{AVL}(h, (d_1, H_1, f_1), (h_\ell, h_r, [(d_i, H_i, f_i)]_{i=2,\dots,k}, seq_h)) = \mathsf{true}$; and
— $k \leq k' + 1 \leq H + 1$, and $k \neq k' + 1$ implies $H_1 = 0$, and $H_k = H$ and
— $k \leq k' + 1 \leq H + 1$, and $k \neq k' + 1$ implies $H_k = H$, and if $H_1 = 0$ and $f_1 = 0$, then we have $h_\ell = h_r = \mathsf{null}$; if $H_1 = 1$ and $f_1 = 1$, then we have $a_{k-1} = r$ and $h_r = \mathsf{null}$; if $H_1 = 1$ and $f_1 = -1$, then we have $a_{k-1} = \ell$ and $h_\ell = \mathsf{null}$; and
— for all $j \in \{1, \dots, k-1\}$, $d_j \mathcal{R} d_{j+1}$ is equivalent to $a_{k-j} = \ell$, and $d_{j+1} \mathcal{R} d_j$ is equivalent to $a_{k-j} = r$.

Intuitively, the proof for a random checking is the proof of presence of a node. The above definition defines how to verify the random checking based on a given position (selected by the verifying user). Loosely speaking, given the proof $(h_\ell, h_r, seq_d, seq_h)$ that (d_1, H_1, f_1) is in the AVL LexTree whose hash value is h and height is H, a user needs to first verify that whether the length of seq_d is equal to the length of seq_h. If they are equal, then the user verifies that proof $(h_\ell, h_r, seq_d, seq_h)$ that (d_1, H_1, f_1) is in the AVL LexTree of hash value h. Otherwise, the user needs to verify one more proof that $H_1 = 0$ and the if $seq_d = [(d_i, H_i, f_i)]_{i=\{2,\dots,k\}}$, then $H_k = H$. This is the check for the case that a user selected a position where the node does not exist. So, in this case, the random checking will check the position which is the longest prefix of the selected position, and there exists a node at this position. Moreover, the verifying user also needs to check the order of all nodes storing the data in $(d_1, H_1, f_1) :: seq_d$.

LEMMA C.6. *Let T be an AVL LexTree of height H and hash value h.*

(*1*) *For all data d in T, there exists $w \in \{\ell, r\}^*$ and $pr = (h_\ell, h_r, seq_d, seq_h)$ such that $seq_d = (d, H_1, f_1) :: seq'_d$ and $\mathsf{Rand}\exists_{AVL}(w, H, h, pr) = \mathsf{true}$, for some H_1, f_1 and seq'_d.*
(*2*) *For all $w \in \{\ell, r\}^*$, if $|w| \leq H$ then there exists a unique pr such that $\mathsf{Rand}\exists_{AVL}(w, H, h, pr) = \mathsf{true}$.*

D. PROPERTIES ON WELL FORMED LOGS

In this section, we show properties on well formed logs to support the proof of our security analysis by using TAMARIN prover.

LEMMA D.1. *Let S be a well formed certificate log. Let S' be a chronological data structure. If $\mathsf{content}_c(S')$ is an initial subsequence of $\mathsf{content}_c(S)$ then S' is a well formed certificate log.*

PROOF. According to Definition 3.22, S being a well formed certificate log implies that $\mathsf{content}_c(S)$ can be denoted by $[\mathsf{h}(req_k, n_k, dg_k)]_{k \in \{1,\dots,N\}})$ where $N = |\mathsf{content}_c(S)|$, for some $req_k \in \mathsf{Req_c}$, $n_k \in \mathbb{N}$, $dg_k \in \mathcal{D}gt_c$ for all $k \in \{1, \dots, N\}$. Moreover, it implies that for all $k \in \{1, \dots, N-1\}$, $N_k \leq N_{k+1}$ and reg_k is apply on dg_k into dg_{k+1}. Thus, if $\mathsf{content}_c(S')$ is an initial subsequence of $\mathsf{content}_c(S)$, we directly have that for all $k \in \{1, \dots, |\mathsf{content}_c(S')| - 1\}$, $n_k \leq n_{k+1}$ and reg_k is apply on dg_k into dg_{k+1}. S' is a well formed certificate log. $\square$

LEMMA D.2. *Let S be a well formed mapping log. Let S' be a chronological data structure. If $\mathsf{content}_c(S')$ is an initial subsequence of $\mathsf{content}_c(S)$ and there exists $t \in \mathbb{N}$, $dg_s, dg_{bl}, dg_r, dg_i \in \mathcal{D}gt_o$ and a bitstring p of size $O(\log(|\mathsf{content}_c(S')|))$ such that*

$$\mathsf{VerifPoP}_c(\mathsf{digest}_c(S'), \mathsf{h}(\mathsf{end}, t, dg_s, dg_{bl}, dg_r, dg_i), |\mathsf{content}_c(S')|, p) = \mathsf{true}$$

then S' is a well formed mapping log.

PROOF. Let us denote $N' = |\mathsf{content}_c(S')|$. According to Definition 3.22, S being a well formed certificate log implies that $\mathsf{content}_c(S)$ can be denoted by $[\mathsf{h}(req^k, t^k, dg_s^k, dg_{bl}^k, {dg_r}^k, dg_i^k)]_{k\in\{1,\dots,N\}}$ where $N = |\mathsf{content}_c(S)|$, for some $req^k \in \mathsf{Req_m}$, $t^k \in \mathbb{N}$, $dg_s^k, dg_{bl}^k, {dg_r}^k, dg_i^k \in \mathcal{D}gt_o$ for all $k \in \{1,\dots,N\}$. Moreover, it implies that $req^N = \mathsf{end}$ and for all $k \in \{1,\dots,N-1\}$,

(1) if $req^k \neq \mathsf{end}$ then $req^k \leq_r req^{k+1}$ and req^k is applied on $(t^k, dg_s^k, dg_{bl}^k, {dg_r}^k, dg_i^k)$ into $(t^{k+1}, dg_s^{k+1}, dg_{bl}^{k+1}, {dg_r}^{k+1}, dg_i^{k+1})$; and
(2) if $req^{k+1} = \mathsf{end}$ then there exists $q \in O(\log(\mathsf{size}_o(\mathsf{digest}_o(dg_s^k))))$ such that for all $j \in \{1,\dots,2^q\}$, there exists values p, dg, dg', a certificates $cert$, a signing keys sk and $t, n, n' \in \mathbb{N}$ such that $n \leq n'$, $\mathsf{id}(cert) = \mathsf{id}(cert')$, $\mathsf{pk}(sk) = \mathsf{key}(cert)$ and:

$$\mathsf{RandM}_O((cert, \mathsf{sign}_{sk}(n, dg, t)), (cert, \mathsf{sign}_{sk}(n', dg', t^{k+1})), dg_s^k, dg_s^{k+1}, p, j, 2^q) = \mathsf{true}$$

Since $\mathsf{content}_c(S')$ is an initial subsequence of $\mathsf{content}_c(S)$, we trivially have that these two properties are true for all $k \in \{1,\dots,N'-1\}$.

Thus, it remains to show that $req^{N'} = \mathsf{end}$. By hypothesis we know that there exists $t \in \mathbb{N}$, $dg_s, dg_{bl}, dg_r, dg_i \in \mathcal{D}gt_o$ and a bitstring p such that

$$\mathsf{VerifPoP}_c(\mathsf{digest}_c(S'), \mathsf{h}(\mathsf{end}, t, dg_s, dg_{bl}, dg_r, dg_i), N', p) = \mathsf{true}$$

By Definition 3.1, we deduce that $\mathsf{h}(\mathsf{end}, t, dg_s, dg_{bl}, dg_r, dg_i)$ is the ${N'}^{\text{th}}$ element of $\mathsf{content}_c(S')$ and so that $req^{N'} = \mathsf{end}$. Therefore the result holds. □