On the impossibility to forge illegitimate proofs of membership in Merkle (Patricia) Trees

Jérémie Albert
jeremie@inblocks.io
inBlocks
Bordeaux, France

Serge Chaumette
serge.chaumette@labri.fr
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
Talence, France

ABSTRACT

The value of an asset often depends on the capability we have to demonstrate its existence at a certain date. Blockchains provide such a service but they tend to be expensive if used intensively. Multilevel ledgers exist that make is possible to cache the operations and thus to reduce the cost of use but not many (if any) have provided proofs that they do not put at risk the assets they register. Therefore the goal of this paper is to demonstrate in an understandable and convincing manner that the so called Precedence multilevel ledger that we have developed at inBlocks does not impair the assets entrusted to it.

1 INTRODUCTION

Assets, in a business perspective, are a promise to a future benefit. They might be photographs, movies, sounds, text documents or structured key/value representations (*e.g.* XML or JSON files). Whatever their format all these information can be considered as binary data and in the following we do not assume any specific format regarding these digital assets; we definitely consider them as binary data and we might refer to them simply as data.

The value of an asset and thus of the **digital asset recording** process depends on the capability we have to demonstrate its existence at a certain date.

The data required to demonstrate such existence is called a **proof of existence**. The keystone value of a proof of existence is its creation timestamping and the capacity of our platform to demonstrate, in a undeniable way, the truthfulness of this timestamp (*i.e.* that it has not and that it cannot be forged). To demonstrate the existence of a piece of data at a certain point in time we usually assume that the recording of its **fingerprint** (a series of bits that is unique for any numeric item) as a digital asset in a secure and distributed ledger like Bitcoin or Ethereum[9], is perfectly safe, meaning it cannot be forged.

inBlocks[3] is a blockchain[4, 8] startup company founded in 2018 that created a SaaS platform designed to record and manage digital assets.We have developed additional layers for the purpose of making the use of a blockchain less expensive and more flexible for our users, and this is our value. Indeed, for real use case scenarios, the cost (in terms of euros or dollars) of this recording is very high. Our goal at inBlocks is to make this process affordable, even for assets whose value might be very low. The solution we provide is a way for our users to minimize their costs by aggregating a number of operations by using **an open-source solution developed at inBlocks called Precedence**.

Precedence is an open-source software solution to certify and prevent forging numerical assets. It is integrated within our platform to allow our customers to create data repositories structured as a blockchain. It has the same intrinsic characteristics, like the impossibility to remove a record from an already existing block without having to recompute all the blocks that refer to it (*i.e.* all the blocks that were created afterwards). The data structure that is used in Precedence is a a Merkle Patricia Tree (MPT for short in what follows). A MPT is a tree data structure designed to reduce the size of the proof of existence of every piece of data that it stores.

The goal of this paper is to demonstrate that the Precedence inBlocks software layers do not impact the security of the overall system. *i.e.* that the security is the same when using inBlocks platform and when directly using a blockchain. This demonstration must inspire confidence: this is the goal of the work presented in this paper to lay understandable and solid foundations to support this confidence.

To achieve this goal we proceed as follows:

(1) we give clean definitions of the different forms of trees that are used (Trees, Prefix Trees or Tries, Merkle Trees and Merkle Patricia Trees),
(2) we then show that no proof of membership of a digital asset secured using such structures can be forged.

By doing so, we hope to advance the way confidence can be explained, described and proven simply so as to be convincing for the users of our platform and of similar registries.

2 TREES, PREFIX TREES AND PATRICIA TREES

2.1 Definitions

A Tree is the combination of a value (the value of its root) and of a number of children (Trees).

Definition 2.1 (Tree).
Let V be a set of values.
If $value \in V$ then $T = (root\_value = value, children = \emptyset)$ is a Tree over V.
If $value \in V, t_1, t_2, ...t_n$ are Trees over V then $T = (root\_value = value, children = \{t1, t2, \ldots, tn\})$ is a Tree over V.

It should be noted that we do not define the notion of node as can be found in the literature. The reason for that is that we are interested in values that are stored and the notion of node is thus not necessary in what follows. Our approach is value oriented rather than structure oriented, which we believe is an original point of view.

The *Children* of a Tree are defined by construction as follows:

Definition 2.2 (Children).
Let T be a Tree.
$Children(T = (root\_value, children)) = children$ is the set of Children of T. We will also write $T.children$.

Definition 2.3 (Leaf).
A Tree with no child ($children = \emptyset$) is called a Leaf.

Definition 2.4 (Internal Tree).
An internal tree is a tree that is not a leaf.

The Root of a Tree is defined by construction as follows:

Definition 2.5 (Root of a tree).
Let $T = (root\_value, children)$ a tree.
$root(T) = root\_value$ is the root of T. We will also write $T.root\_value$.

Definition 2.6 (Sub Tree).
Let T be a Tree.
S is a Sub Tree of T iff $S = T$ or S is a Sub Tree of a child of T.

Definition 2.7 (Father).
Let T be a Tree and S one of its Sub Trees.
The Father of S in T is the Tree F, if it exists, such that S is a SubTree of T and $S \in Children(F)$, $\emptyset$ otherwise.

Notation: we will write $Father(S \in T)$ to denote the Father of S in T and $Father(S)$ when there is no ambiguity on T.

2.2 Operations on a Tree

The Brothers of a Tree are the Children of its Father minus itself.

Definition 2.8 (Brothers).
Let T be a Tree.
If Father(T)=∅ then Brothers(T)=∅ else Brothers(T)=Children(Father(T))-T.

The Uncles of a Tree are the Brothers of its Father.

Definition 2.9 (Uncles).
Let T be a Tree.
If Father(T)=∅ then Uncles(T)=∅ else Uncles(T)=Brothers(Father(T)).

The i-Uncles of a Tree are its Uncles at the i^{th} level.

Definition 2.10 (i-Uncles).
Let T be a Tree.
i-Uncles(T,1)=Uncles(T).
For $i > 1$, i-Uncles(T,i)=Uncles(Father(T),i-1) if Father(T)≠ ∅, ∅ otherwise.

The *i-Uncles of a Tree are its Uncles and the Uncles of its ancestors in the Tree up to the i^{th} level.

Definition 2.11 (⋆ i-Uncles).
Let T be a Tree.
$\star i - Uncles(T, 1) = Uncles(T)$
For $i > 1$ ⋆ i-Uncles(T,i)=Uncles(T)⋃⋆ i-Uncles(Father(T),i-1) if Father(T)≠ ∅, ∅ otherwise.

2.3 Prefix Trees and Patricia Trees

Prefix Trees (or Trie for reTRIEval of data) and Patricia Trees[6] are optimized trees in terms of the organization of the data they store. They support the efficient insertion and retrieval of these data.

A Prefix Tree (or Trie) is typically used to store strings, each node carrying only one letter. Strings that share a common prefix have the same ancestor in the Trie.

A Patricia Tree is an optimisation of a Trie. It is a Prefix Tree where each leaf that has no brother is iteratively merged with its father (and thus contains a string instead of a single character). It provides a way to access information in a space and time efficient manner (better than what can be achieved with a Trie) as described in the reference paper. Patricia stands for *Practical Algorithm To Retrieve Information Coded In Alphanumeric.*

3 MERKLE TREES ANS MERKLE PATRICIA TREES

3.1 Merkle Tree

A Merkle Tree (MT for short) is a tree built as follows: the value of a leaf is a hash; the value of an internal node (all nodes but the leaves) is the hash of the concatenation of the values of its children. It should thus be noted that the order of the children of a node in a merkle tree matters.

The use of a hash function (more precisely of a one-way and collision-resistant hash function) is the key to the cryptographic features supported by Merkle Trees and Merkle Patricia Trees. By hypothesis, generating the same image from different pieces of data (a collision), is too costly and as such considered impossible. This hypothesis is the keystones of many if not all security systems and this is also the case for the proofs that will follow in this paper.

Rationale for defining Merkle Trees in the context of the blockchain
To check the presence of a value in a MT, it is enough to provide the root of the tree, the values of its brothers, of its uncles, and of the uncles of all its ancestors in direct line. It thus makes it a significant help to assert the existence of a given data asset at a given timestamp. This property will be detailed later in this paper.

3.1.1 Definitions. Let # be a one-way hash function defined over a set of values (digital assets) V. Let $H = \#(V)$ be the set of hash values of the members of V.

Definition 3.1 (Merkle Tree).
If $h \in H$ then $MT = (root = h, \emptyset)$ is a Merkle Tree over H.
If $\{mt_1, mt_2, ..., mt_n\}$ is an ordered set of Merkle Trees over H then $T = (root = \#(Root(mt_1).Root(mt_2)....Root(mt_n)), children =$

$\{mt_1, mt_2, \ldots, mt_n\})$ *is a Merkle Tree over* H *and* $root$ *is the root of this tree.*

It should be noted that the leafs of the tree are the only nodes that carry a value (more precisely a hash) of V (more precisely of $H = \#(V)$).

The following property results from the construction of a Merkle Tree.

Property 3.1. *A Merkle Tree is a Tree.*

All the definitions and operations defined for trees (above) thus also apply to Merkle Trees.

3.1.2 Operations on a Merkle Tree. We introduce the notion of a *Future Merkle Tree* (FMT for short) that will be useful to describe attack attempts that consist of replacing part(s) of the MT with fake data. A FMT is a tree with a hole, *i.e.* a tree where a sub tree has been removed, leaving a hole in the structure. This hole can be later populated with a replacement tree/sub tree.

Rationale for defining FMT in the context of the blockchain Replacing a subtree of a MT is the way an attacker would forge false proofs of membership in the blockchain. We will thus use this notion of FMT to prove the impossibility of this attack. This will be detailed later in this paper.

Definition 3.2 (Future Merkle Tree).
A Future Merkle Tree is a Merkle Tree with a hole. It is noted $FMT[]$.

Definition 3.3 (Populated Future Merkle Tree).
A Populated Future Merkle Tree is a Future Merkle Tree FMT the hole of which has been filled with a Merkle Tree MT. It is noted $FMT[MT]$.

Note that populating a Future Merkle Tree should be done with a Merkle Tree and also implies a recompilation of part of the tree (because the hash values carried by the node above the populated hole depend on the tree used to populate he hole). If not, the result is a Tree but not a Merkle Tree.

Definition 3.4 (Subtree substitution).
Let mt *and* mt' *be Merkle Trees,* $mt \neq mt'$.
Let $MT = FMT[mt]$ *be a Merkle Tree.*
$MT\{mt \rightarrow mt'\} = FMT[mt']$ *is the substitution of* mt *by* mt'.

$MT\{t \rightarrow t'\}$ is the same tree as MT except mt has been replaced by mt' and consequently part of the tree has been recomputed.

3.1.3 Properties of a Merkle Tree.

The following property establishes that is is not possible to change a subtree of a Merkle Tree without modifying its root.

Property 3.2 (One Sub Tree substitution instability).
Let mt, mt' *be two distinct Merkle Trees.*
Let $MT = FMT[mt]$ *be a Merkle Tree.*
Then $root(FMT[mt']) \neq root(MT)$

Proof. The proof is built by induction on the depth d of the tree.

- for d=1
  Let MT be a Merkle Tree of depth 1.
  $MT = (root, \emptyset) = [(root, \emptyset)]$
  Let MT' be another Merkle Tree, and assume that $Root(MT') = root$. Then there are two possibilities:
  (1) either $MT' = (root, \emptyset)$ which is not possible because we said the replacement Tree should be different from the original Tree
  (2) or $MT' == (root, mt_1, ..., mt_n)$ with $root = \#(\#_{i=1}^{n} mt_i)$. # being a hash function, this means that we are in presence of a collision of the hash function. **By hypothesis (see section 3.1), generating such a collision, *i.e.* computing such a set of children is too costly and considered impossible. Consequently, even if this is theoretically possible, this is practically impossible, which is enough from a practical point of view.**
- for $d > 1$
  Let $MT = FMT[mt]$ be a Merkle Tree of depth d, $d > 1$
  By induction hypothesis we have
  $\forall mt' \neq mt, Root(FMT[mt']) \neq root(MT)$
- for $d + 1$
  Let $MT = (\#(mt_1.root, ..., mt_n.root), mt_1, ...mt_n)$ and let mt_i be the tree where the substitution is done (it can be mt_i that is substituted or one of its subtrees). This substitution thus takes place at depth $\leq d$ and produces a new tree mt_i'. Since depth of $mt_i = d - 1$, we have by induction hypothesis $Root(mt_i') \neq Root(mt_i)$ and it results that
  $\#(mt_1.root, ..., mt_i'.root, ..., mt_n.root) \neq$
  $\#(mt_1.root, ..., mt_i, ..., mt_n.root)$

□

We now show that making several substitutions is equivalent to making only one.

Definition 3.5 (Multi Future Merkle Tree). *A Multi Future Merkle Tree is a MT with n holes. It can be written as follow* $MT = FMT_n([]_1[]_2...[]_n)$. *The indexes of the holes will not be indicated when there is no ambiguity.*

Property 3.3 (N Sub Trees substitution instability). *Let* $MT = FMT_n([mt_1][mt_2]...[mt_n])$ *be a Merkle Tree.*
Let $mt_1', mt_2', ..., mt_n'$ *be a set of Merkle Tres with* $mt_j \neq mt_j'$.
Then $root(FMT_n([mt_1'][mt_2']...[mt_n']) \neq root(MT)$

Proof. The proof is straightforward.

- for $n = 1$ we have shown the property earlier in this paper (property 3.2).
- for $n > 1$.
  Let A be the smallest common ancestor of $mt_1, mt_2, ...,$ and mt_n. We thus have $MT = FMT_n([mt_1][mt_2]...[mt_n]) = FMT'([A])$
  In the substitution process A is thus replaced by A', $A' \neq A$. Since the property holds for $n = 1$, we have $FMT'([A]) \neq FMT'([A'])$

□

3.2 Merkle Patricia Tree

3.2.1 Overview. A Merkle Patricia Tree (MPT for short) is a Tree that stores $(path, value)$ pairs, and given a path it is possible to retrieve the associated value (as in a Patricia Tree). It is combined with a data base that contains pairs $(key\_array, value)$ each entry being itself referred to by a key, and key_array containing the key to access each of the children of the current node.

When looking for a node, you use the root key (that must thus be provided in some way) to access the associated entry in the data base and recursively use the key_array index to go down the tree. In other words "*you start by looking up the root hash in a flat key/value DB to find the root node of the trie. It is represented as an array of keys pointing to other nodes. You would use the value at index"* [appropriate index] "*as a key and look it up in the flat key/value DB to get the node one level down.*" [2].

A Merkle Patricia Trie thus "*provides a cryptographically authenticated data structure that can be used to store all (key, value) bindings.*"[2]. It makes it possible to check the existence of a $(path, value)$ pair in a given MPT, giving away less information than what is required in a MT.

To summarize, a MPT can be seen as a combination of a Merkle Tree with a Patricia Tree. The Patricia structure is used to define/locate the storage location of a piece of data in the tree and to accelerate the retrieval process, and the Merkle construction is used to ensure the cryptographic features (using a hash combination process) as in a Merkle tree.

Rationale for defining Merkle Patricia Trees in the context of the blockchain
To check the presence of a value in a MPT, it is enough to provide the root of the tree, the pair (path, value) given at insertion time and what is referred to as a proof which is the series of hashes leading from the leaf to the root of the Tree. It thus makes it, like MPT, a significant tool to assert the existence of a given pair (path, data asset) at a given timestamp, providing much less information than by using a simple MT. This property will be detailed later in this paper.

3.2.2 Definitions. Let V be a set of values (digital assets) and P be a set of paths defined over an alphabet AZ, of size $|AZ| = r$ (we call nibble a small part of a path, and a nibble is itself a path). Let # be a one-way hash function over V. Let $H = \#(V)$ be the set of hash values of the members of V.

Definition 3.6 (r-Merkle Patricia Tree).
CUT is a r-Merkle Patricia Tree, and CUT.key=0.
If $path \in P$ *and* $value \in V$, *then* $T = (node = (nibble = path, value = value, children = \emptyset), key = \#node)$ *is a r-Merkle Patricia Tree over* (P, V).
If $path \in P$, $v \in V$ *and* $\{mpt_1, mpt_2, ..., mpt_r\}$ *is an ordered set of r-Merkle Patricia Trees over* (P, V) *then* $T = (node = (nibble = path, value = v, children = \{mpt_1.key, mpt_2.key, \ldots, mpt_r.key\}), key = \#node)$ *is a r-Merkle Patricia Tree over* (P, V)

The above definition makes it clear why these trees are called Merkle Patricia Trees: like for Merkle Trees, the root is a *fingerprint of the entire data structure*[1].

We now define the path and the proof of a MPT.

Definition 3.7 (path).
Let T be a MPT.
if Father(T)=$\emptyset$ $path(T) = \epsilon$
else $path(T) = path(Father(T)).(T.path)$.

The path is either provided at insertion time or built at insertion time based on a serialisation of $T.value$. If a path p is provided at insertion time, the insertion process should be so that $path(T) = p$.

Definition 3.8 (proof).
Let T be a MPT.
if Father(T)=$\emptyset$ $proof(T) = \epsilon$
else $proof(T) = proof(Father(T).(T.key)$.

The proof is constructed and returned at insertion time or can be obtained by providing the path of a node.

4 PROOF OF MEMBERSHIP

As explained earlier in this paper, a proof of membership of a data asset is itself a piece of data (globally called a proof) than can be used to prove that this data asset has been recorded in a ledger at a give time (at a given time stamp).

The question is then: is it possible to forge a proof that would pass the verification process to claim that a piece of data has been inserted in a ledger at a given timestamp?

4.1 Proof of membership in a Merkle Tree

In a Merkle Tree (MT) a proof of membership is composed of :

- the digital asset (indeed a hash of a digital asset) stored in a leaf of the tree
- the Brothers of this leaf
- the Uncles of this leaf and of all its ancestors (in direct line)
- the root of the tree

If we use the notations and definitions that we have given in this paper, these are:

- the value of a Leaf L=(value=#(*digital asset*), 0)
- Brothers(Leaf)
- *-Uncles(Leaf)
- Root(Tree)

The algorithm used to check the validity of the proof of membership can the be expressed as follows :

```
1  Bool verify(Node_Value, Node_Brothers,
2              Node_*-Uncles, Tree_Root) {
3    if ((Node_Brothers==0) and (Node_*-Uncles==0))
4      then /* we have reached the Root of the Tree)
5         if (Node_value == Tree_Root.value) then
6           return True
7         else
8           return False
9
10   /* Compute what should be the value
11      of the Father of the current Node */
12   expectedFatherValue =#(Node_value..B∈Node_Brothers B.value)
```

```
13
14   /* Retrieve brothers of father */
15   Father_Brothers= [...] /* from Node_*-Uncles */
16   /* Retrieve uncles of father and of its
17      direct ancestors */
18   Father_*Uncles= Node_*-Uncles - Father_Brothers
19   /* Verify that the expected Father Value is part
20      of the Merkle Tree at hand */
21   return verify(expectedFatherValue, Father_Brothers,
22                 Father_*-Uncles, Tree_Root)
23 }
```

In the above algorithm the values of all the nodes of the Tree on the path from the leaf to verify, up to the root of the Tree, are reconstructed. The only way to fool this verification process would be to do so that the root that is eventually reconstructed in this process starting from a substituted sub-tree equals the initial root of the tree at hand. We have shown (properties 3.2 and 3.3) that this is not possible.

4.2 Proof of membership in a Merkle Patricia Tree

In a Merkle Patricia Tree (MPT) a proof of membership is composed of:

- a pair itself composed of:
  - the path (definition3.7) given at insertion time or obtained from the insertion process. Paths are sometimes referred to as keys (for insertion keys), but we find it confusing with respect to the hashes that are used as access keys
  - the value associated with the path
- the proof (definition 3.8) associated with the (path, value) pair

To describe the proof verification algorithm we require an additional function to navigate between hashes and the nodes ot the MPT.

We thus define the function *child* such that

Definition 4.1. *child*(*hash*, *child_hash*) *returns the node, the hash of which is child_hash.*

In effective implementations, this function is implemented by means of a database associated with the tree.

The algorithm used to check the validity of the proof of membership can now be expressed as follows (a version taking into account the Ethereum optimization of a MPT can be found in [7]):

```
1  Bool verify(root_hash, target_proof, proof_idx,
2              target_path, path_idx,
3              target_node_value) {
4
5    current_node=child(root_hash,target_proof[0])
6
7    nibble_length=len(current_node.nible)
8
9    if (path_idx==0)
10    then
11       /* checking root */
12       if (target_proof[0]!=root_hash)
13         then
14           return False
15    else
16       /* for all nodes but root check that
17          the series of # remains valid */
18       if (target_proof[proof_idx]!=root_hash))
19       then
20         return False
21
22   if (path_idx + nibble_len < length(target_path))
23   /* we are at the level of an internal node */
24     if (current_node.nibble==
25       target_path[path_idx:path_idx+nibble_len])
26       then
27         /* path corresponds */
28         new_expected_root=Node_proof[proof_idx]
29         return verify(
30           new_expected_root,
31           target_proof, proof_idx + 1,
32           target_path, path_idx + nibble_len,
33           target_node_value)
34
35   /* we are at the level of a leaf */
36   /* check if both nibble and value match */
37   if ((current_node.nibble==
38     target_path[path_idx:path_idx+nibble_len])
39     and
40     (current_node.value=target_node_value))
41     then
42       return True
43
44   /* none of the tests has been successful */
45   return False
46 }
```

We have shown that forging a proof for a given value that passes the verification process is not feasible for MT (see section 4.1). The verification process being almost the same for MPT, it is also not feasible: instead of starting the verification from a leaf of the tree, the verification starts at the root. Still, the reasoning is the same. The root hash can only be the valid root of a given path provided the components of the path are those that have been used to build the root. Fooling the proof verification process is thus impossible.

Since the proof associated to a given value cannot be forged, descending the tree also makes it possible to verify the path associated to the target value. The path and the proof should describe the same navigation in the tree when descending it.

By doing so it is thus possible to verify the existence of (path, value) in the considered tree.

5 PRECEDENCE: A PROOF OF SAFETY

Our proof addresses a theoretical model of the architecture (at the level of its design) rather than its effective implementation that would require effective code analysis, which is currently out of the scope of our work.

5.1 Architecture of Precedence

Precedence is a software written in both NodeJS (this version of Precedence is available on GitHub[1]) and Java (not yet released). It relies on the implementation of the Merkle Particia Trie that is used in Ethereum[2] and that is available on GitHub[3]. Most blockchain technologies are inefficient in terms of number of transactions (*i.e.* number of writes) per second. For example Bitcoin supports in average a maximum of 7 writes per second[5] and Ethereum around 15[5]. The main purpose of the Precedence stack is to provide a data storage mechanism that scales in term of number of possible writes per second without tampering with the security of data. The way we make it scale is by adding a multi-level mechanism so that we can create a sort of forest of Precedence instances that act as independent data layers. Each layer (except the top level one that we call Layer 1) periodically writes information in the upper level layer. In our architecture we assume that Level 1 is a distributed blockchain like Bitcoin or Ethereum. All the other layers (from 2 to n) might not be public, and this privacy must not lead to a lack of confidence regarding the proof-of-existence that this software stack generates. The aim of this paper is to show that whatever the number of layers we use, the proof-of-existence of a piece of data stored in a layer n that is run by the Predecence stack is as secured (*i.e.* cannot be tampered with) as a proof-of-existence of a piece of data that would be stored directly into a Layer 1 ledger.

5.2 Specific implementation information

Every piece of data sent to Precedence contains both a key and a value. The key is the document identifier that is guaranteed unique inside the system (at this Layer) and the value is a blob (any binary data). This information is encapsulated in a JSON document.

We now give some definitions that will help explain the Precedence internal data structure:

- *seed* (that could have been named salt but as we use it to obfuscate in the same way all the values of a document we could argue that it is both a seed and a salt as defined most of the time in the literature) is a random and server-side computed number represented as a UUID (that is not time-dependent and assumed randomly generated like the UUIDv4)
- $hash(d)$ is the value resulting from the hash computation of the data d. We assume that all the hash function used are cryptographic (and so one way) hash functions.
- $obfuscated(d, seed) = hash(concat(hash(d), seed))$ is a function used to obfuscate any piece of information (d in this example)

Each Precedence record is built from a *provable* document named D in the following, that contains at least 3 key/value items defined as follows (we assume here the asset identifier is id and its associated data are d):

- $provable.id = id = hash(key)$, where key is the asset id
- $provable.seed = obfuscated(seed, seed)$
- $provable.hash = obfuscated(d, seed)$

In the Merkle Patricia Trie used in Precedence, the key (or path) we use is $provable.id$ (= id) and the value is $hash(D)$.

5.3 Proof of safety

Following the Merkle Patricia Trie structure, we can demonstrate easily that a document (key, value) is part of it and so part of the Precedence data layer, whatever its level.

Each time a block is generated in a Precedence Layer n ($n > 1$), its root hash is written as a new document in a Precedence Layer $n-1$. We can then demonstrate the existence of the block of the Layer ^{n} in the Precedence of the Layer $n-1$. The above construction builds a forest of Precedence instances, that is by construction structured as a Merkle Patricia Tree. Based on the work presented in this paper we can then recursively demonstrate that the existence of any given piece of information written using Precedence can be asserted in the Layer 1 ledger (for now we use Ethereum and a dedicated Smart Contract to store those values).

6 CONCLUSION

In this paper we have shown that the multilevel Merkle Patricia Trees used in Precedence are simple data structures which allow building proof-of-existence that cannot be tampered with (as long as the one-way cryptographic function it relies on can neither be tampered with).

The open-source Precedence stack was not designed to be distributed among a set of participating nodes. However, we are currently working on new data distribution features that could bring a peer-to-peer autonomous confidence (to make the system independent from a distributed Level 1 ledger) still preserving a by-design data privacy capability. We are convinced that this proof-of-existence feature can be achieved in a distributed way without any global (and often costly) consensus mechanism which is required in blockchain solely to prevent the double-spending issue.

[1] https://github.com/inblocks/precedence
[2] https://ethereum.org/
[3] https://github.com/ethereumjs/merkle-patricia-tree

REFERENCES

[1] 2016. How does a Merkle-Patricia-trie tree work? https://ethereum.stackexchange.com/questions/6415/eli5-how-does-a-merkle-patricia-trie-tree-work. Accessed: 2023-10-20.
[2] Ethereum contributors. 2023. Merkle Patricia Tree. https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/. Accessed: 2023-10-20.
[3] inBlocks team. 2023. inBlocks Web Site. https://inblocks.io/. Accessed: 2023-10-18.
[4] Lorne Lantz and Daniel Cawrey. 2020. *Mastering Blockchain: Unlocking the Power of Cryptocurrencies, Smart Contracts, and Decentralized Applications.* O'Reilly Media, Inc.
[5] Debasis Mohanty, Divya Anand, Hani Moaiteq Aljahdali, and Santos Gracia Villar. 2022. Blockchain Interoperability: Towards a Sustainable Payment System. *Sustainability* 14, 2 (2022). https://doi.org/10.3390/su14020913
[6] Donald R. Morrison. 1968. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM (JACM)* 15 (1968), 514 – 534. https://api.semanticscholar.org/CorpusID:3335450
[7] Pierre-Alain Ouvrard. 2019. Merkle proof verification for Ethereum Patricia tree. https://ouvrard-pierre-alain.medium.com/merkle-proof-verification-for-ethereum-patricia-tree-48f29658eec. Accessed: 2023-10-20.
[8] Melanie Swan. 2015. *Blockchain.* O'Reilly Media, Inc.
[9] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151 (2014), 1–32.