TODA POP Structure

Version 1.414
Team TODA

October 2019

1 Introduction

This document describes the TODA Protocol's Proof Of Provenance (POP) data structure. In particular, it provides the data layouts for version one of the POP parts, which are used when hashing those parts for inclusion as well as at rest and on the wire. It serves as the canonical source and normative specification for all V1 POP structures.

In addition to the concrete data structure descriptions, this document also includes brief descriptions of the algorithms for constructing certain portions of that data structure, an abstract description of and motivation for particular pieces, and proofs of the integrity guarantees provided by the POP structure.

The components of the POP structure are presented here starting with the chronologically first piece, the file kernel, and then building up from the file detail, to the file trie, the transaction packet, and finally the cycle proof. This is the way each POP slice is actually built, from the bottom up.

When analyzing a POP slice in a proof context, however, it is customary to start with the cycle root at the top and work down. This is the direction of decomposition while proving the structural properties of the POP in the later portion of this document.

Within this document the word 'hash' refers to application of the cryptographic hash function SHA256, or the result of said application, or a 32 byte string, depending on context. The word 'null' used in a hash context refers to the 32 byte string of zero bytes (aka the null hash).

2 Structure Descriptions

2.1 File Kernel

A file is defined by a data structure consisting of five hashes concatenated together. This is known as the file's *kernel*, and the hash of the kernel is known as the file's *identifier* (or id).

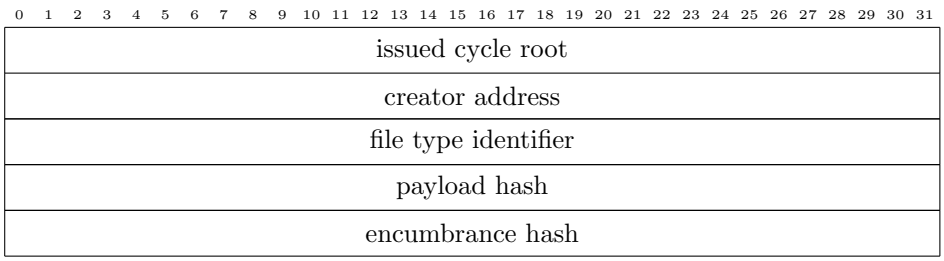

Figure 1: The *file kernel*

The *issued cycle root* is the current cycle root at the time of the file's creation. For a file to be valid its initial transaction **MUST** take place in the immediately following cycle.

The *creator address* is the address of the creator of this file. For a file to be valid, the creator address **MUST** match the address of the initial transaction.

The *file type identifier* is a file id or null. That is, a file's type is either the id of another file, which is designated as the type of this file, or it is null if this file has no type. For a file `f` to be valid its file type **MUST** either be null, or a file type file that was valid during `f`'s issued cycle root.

The *payload hash* is the hash of any arbitrary bytes that make up the initial content of this file. There are no restrictions on this, and it is allowed to be null. It is perfectly acceptable to re-use a file's payload in a different file.

The *encumbrance hash* is the hash of a proper encumbrance packet. The encumbrance packet restricts instances of this file type (that is, files which use this file as their type). Those restrictions are not within the scope of this document. For a file to be valid its encumbrance hash **MUST** either be null or the hash of a proper encumbrance packet.

The file's kernel is globally unique. As a corollary each file has a unique *file identifier*, or file id, which is the hash of its kernel. Section 3 proves this as one of the fundamental properties of the POP structure.

2.2 File Detail

The file detail contains updates intended to be made to a file. It is defined as the concatenation of the following three hashes.

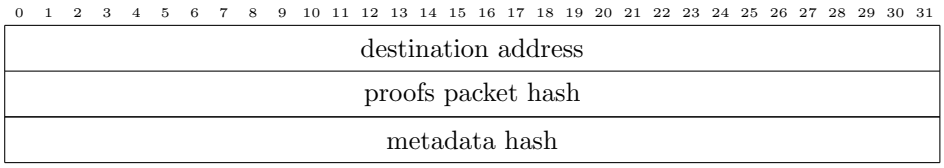

Figure 2: The *file detail*

The *destination address* is the address of the intended recipient of this file. If this file detail is part of the file's POP, and there is no more recent file detail in the file's POP, then the destination address is said to own the file. If the destination address is null then this file is considered *invalid* beginning in this cycle.

The *proofs packet* contains proof that this file detail entry met any requirements that were put on it by encumbrances or operations, if any such requirements exist. If no such requirements exist then the proofs packet hash should be null.

The *metadata hash* is the hash of arbitrary metadata to be associated with this file in this cycle. If there is no metadata to associate with this file, this should be the null hash. There are no restrictions on this field, and like the payload field the protocol never examines it.

Note that the file detail does not need to be unique: many files in the same transaction, and indeed across transactions, may all share the same file detail hash. In such cases if uniqueness is desired one may hash a nonce value for the metadata hash field.

A given file may have at most one file detail per cycle in its POP. This is proven in Section 3 as one of the POP guarantees.

2.3 File Trie

The file trie is a Merkle trie which associates file ids with file detail hashes. A *file proof* is the Merkle proof associating a single file detail with a file id. The file detail associated with a file's id by its owner's file trie is the only valid representation of operations on that file.

The file trie for a given address in a given cycle contains all of the operations performed over files owned by that address in that cycle. Files owned by that address that are not present in that file trie for that cycle are said to have *null proofs*. Because the file detail referenced in the file proof is the only valid representation of operations on a file, a file with a null proof is demonstrably not operated on within the context of the file trie.

2.3.1 File Trie Wire Format

The file trie is constructed using an efficient trie building algorithm described below. One effect of this is the simplicity and compactness of the on-the-wire format for file tries and file proofs.

The file trie is the concatenation of a number of frames, where each frame is the concatenation of the following two hashes:

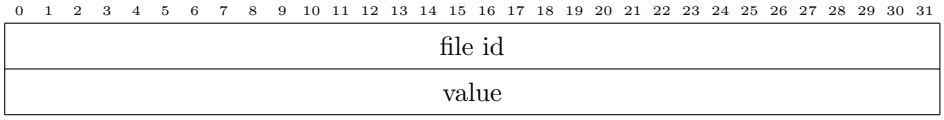

Figure 3: File trie frame

When sent over the wire, the client **SHOULD** send the frames in Lusin–Sierpiński-Kleene–Brouwer order by file id.

2.3.2 File Trie Hashing

The file trie is constructed locally, and may contain a very large number of files. It therefore requires an optimized Merkle trie building algorithm.

Consider the following tree:

```
File-ID  Value
0000     V
0001     W
0010     X
0100     Y
0101     Z
```

To achieve consistent Merkle trie building, we employ the following algorithm:

1. Sort the rows by file id
2. Find two adjacent rows whose bit strings share the longest common prefix, e.g. `File1` and `File2`
3. Merge those two rows, resulting in a single row that associates the lesser of the two file ids with `H(File1,Val1,File2,Val2)`
4. Repeat steps two and three until a single file-value pair remains
5. Finally, calculate `H(File,Value)`

For the example above,

- First 0000 and 0001 would be merged, resulting in `0000 -> H(0000V0001W)`
- Then 0100 and 0101 would be merged, resulting in `0100 -> H(0100Y0101Z)`

- Our table now looks like

```
File-ID  Value
0000     H(0000V0001W)
0010     X
0100     H(0100Y0101Z)
```

- Then 0000 and 0010 would be merged, resulting in
  `0000 -> H(0000H(0000V0001W)0010X)`
- Next 0000 and 0100 would be merged, resulting in
  `0000 -> H(0000H(0000H(0000V0001W)0010X)0100H(0100Y0101Z))`
- Finally, the root hash of the Merkle trie is
  `H(0000H(0000H(0000H(0000V0001W)0010X)0100H(0100Y0101Z)))`

The hash of the final pair is called the *file trie root*, and is a component of the transaction packet.

Note that because file ids are sent and stored ordered, step one can typically be skipped, saving a factor of $O(N \log(N))$ work during the setup.

Note also that step two can be performed on local maxima rather than requiring a global maximum, changing this algorithm from $O(N^2)$ time to $O(N)$ time.

2.4 Transaction Packet

The transaction packet packages together the file trie root, the current cycle root with the address and signature. The hash of this packet is incorporated into the cycle trie as the value of this address.

The transaction packet is the concatenation of the following four hashes:

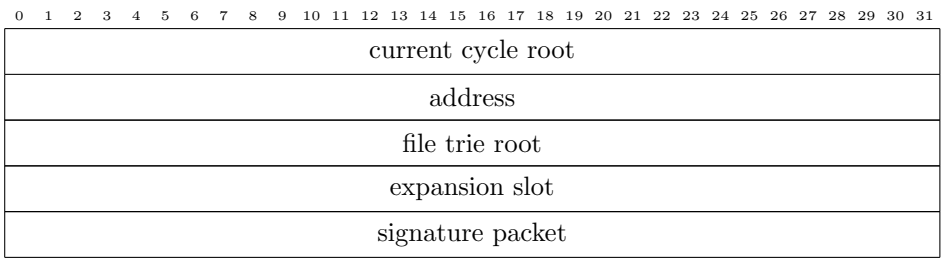

Figure 4: The *transaction packet*

The current cycle root is required to prevent replay attacks.

The transaction packet (or txpx) is the only place the current cycle root and signature appear for normal file transactions. This means you can prepare file tries ahead of time, which is helpful if you have a large number of files.

The *expansion slot* **MUST** be the null hash.

The first four hashes of the transaction packet, consisting of the current cycle root, address, file trie, and expansion slot **MUST** be validly signed.

Please see the address specification file for details on addresses and signature packets.

2.5 Cycle Trie

The cycle trie is a prefix-encoded Merkle trie which associates addresses and hashes of transaction packets. A *cycle proof* is the Merkle proof associating an address with a transaction packet hash. The address contained in a transaction packet, and the address associated with the hash of that transaction packet must agree. If they do not, the address is treated as though it is not present.

The cycle trie for a given cycle contains all the address/txpx pairs for that entire cycle. Addresses that are not present in that cycle are said to have a null proof, and the cycle trie contains those implicitly as well.

It is worth noting that the txpx is associated with a given address (by the cycle trie), and must be signed by that same address. Additionally, any operation performed must be included under a txpx that is signed by the address performing it. Thus we have that no operation can be taken by an address without a valid txpx referenced via the cycle proof for that address. Consequently, if there is demonstrably no cycle proof for an address in a cycle, it constitutes a proof that the address did not perform any operations in that cycle. This is the cycle trie version of a null proof

The data structure for encoding a cycle proof is defined as follows:

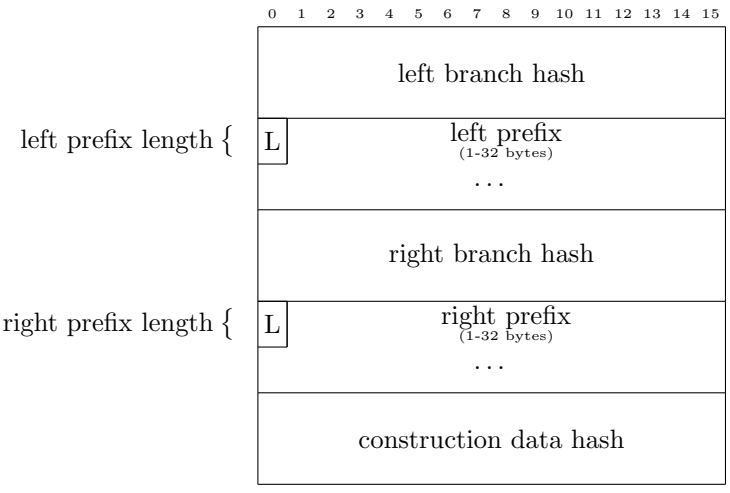

Figure 5: Cycle proof frame (prefix blocks not necessarily word-aligned)

- The left and right *prefix length*, denoted as `L` in figure 5, are one byte unsigned integers that are one less than the prefix length in bits. Thus `0x00` is a length of 1, and `0xFF` is a length of 256 bits.

- The left and right `prefix` are left-padded to the nearest byte boundary. The number one is represented as `0000 0001` (in binary).
- The length of the `prefix` field in bytes is equal to $\lceil \textit{prefix length}/8 \rceil$
- The left prefix `MUST` start with a 0. The right prefix `MUST` start with a 1.
- The *construction data hash* is null in v1 except for the signature of the gardener at the topmost branch, just below the cycle root.

Cycle proof The value of a branch is defined recursively as `H(frame)` where the frame is defined as above, including both of the subbranch hashes and the construction data hash. When a leaf is reached, the txpx value is used instead.

Figure 6: Cycle proof assembled from varied-length frames

2.5.1 Null Cycle Proofs

The cycle trie encodes nulls in two ways: explicitly or implicitly.

- The cycle trie can explicitly encode that a particular branch is null by specifying the *value* of that branch to be null (32 bytes of zeroes).
- The cycle trie can implicitly encode that a branch is null by compressing it into a branch prefix. A branch prefix of `0101` (with length 4 bits) specifies:
  - This left hand branch branched at `0`, as expected (and the right hand sibling must start with `1`).
  - There are no branches starting with `00`, `011`, or `0100` in this cycle trie. They have been skipped, and any addresses contained within those branches are provably null.

This means that if an address does not participate in a cycle, the cycle proofs of the closest address to its left or right which *did* participate serve as the null proof of that address.

2.5.2 Forced Branch at Bit One

For implementation reasons, we force a branch at the first bit in the cycle trie. This means that if all of the addresses which participated in a cycle start with 0, or they all start with 1, then we encode an *explicit* null into the other branch.

Example: Suppose the only address in a cycle started with `0x7`. The frame created would have the following two branches:

- Left: prefix length of 1 bit, prefix of `0`, and a value of null.
- Right: prefix length of 256 bits, prefix exactly matching the entire address, and a value of the address's txpx.

3 Proofs of Ownership Properties

The POP structure provides a foundation for creating and transferring unique digital assets. As such it is necessary to demonstrate that this structure upholds the expected properties, the most important of which is that any given file has at most one owner at one time.

To do this we first we need to define what constitutes a valid POP (and any POPs discussed will be expected to be valid unless stated otherwise). For this definition, we start with the definition for a *POP slice for a file having a file id of* `id`*, belonging to* `a` *in cycle* C_k as consisting of:

- The cycle hash C_k
- A Merkle proof associating the (possibly null) value p with the address `a` in the cycle trie whose root hash is C_k (*the cycle proof*)
- If p is not null, the txpx whose hash is equal to p (*the txpx*)
- If p is not null, A Merkle proof associating the (possibly null) value `f` with the file id `id` in the file trie whose root hash is the file trie root in the txpx (*the file proof*)
- If `f` is not null (and implicitly, p is not null), a file detail whose hash is equal to `f` (*the file detail*)

In the cases where the POP slice does not have a valid file detail (which is equivalent to it not containing a destination address), the POP slice forms a *null proof*.

We then define a POP for a file id `id` with respect to a sequence of cycles $C = \{C_0, ..., C_n\}$ recursively as follows:

If the issued cycle root in the file kernel that hashes to `id` is $C_n - 1$, and the creator address in that same file kernel is $\mathtt{a}_0$, then the POP is simply the POP slice for the file id `id`, belonging to $\mathtt{a}_0$ in C_n. In order for the file to exist, this POP slice must not be a null proof.

If the issued cycle root in the file kernel that hashes to `id` is C_k for $0 \leq k < n-1$, then the POP for `id` in C is defined to consist of the POP for `id` in $\{C_0, ..., C_{n-1}\}$ (which we'll call POP'), and the POP slice for `id`, belonging to $\mathtt{a}_n$ in C_n. The value for $\mathtt{a}_n$ (*the owner at the end of* C_{n-1} *or at the beginning of* C_n) is taken to be the destination address in the file detail from the most recent POP slice in POP' which is not a null proof (where by most recent, we mean being for the cycle C_i with the largest satisfactory value of i).

This definition leaves us to show that given a sequence of cycle roots and a file id, that file has a single minimal POP with respect to that sequence. This is proved using structural induction on the POP itself, starting from that file's first POP slice. Proving the uniqueness of ownership follows as an easy corollary.

Each POP slice represents either the creation of the file, a transaction for that file (which may involve a change of ownership, or maintain the current

owner), a null proof showing that nothing happened to that file in that cycle, or the burning of the file.

We rely on standard cryptographic assumptions, including the collision resistance of cryptographic hash functions.

We define a *minimal POP* as a POP containing fully trimmed Merkle proofs for each slice.

Note that the cycle trie and file trie must have the important properties of a well-constructed Merkle trie. In particular, we require they be uniquely represented by their root hash and return at most one value for any lookup key in their intended search space.

Theorem 1. *Given a file* `f` *with id* `id` *and a sequence* C *of cycle roots* $\{C_0, C_1, ..., C_n\}$ *which contains* `f`*'s cycle of creation, then* `f` *has a single minimal POP with respect to* C*.*

Proof. Consider a file `g` whose file id is also `id`. We will demonstrate that the minimal POP of `g` over C, POP^g, is necessarily identical to POP^f, the minimal POP of `f` over C. As a result, there is at most one minimal POP that is valid for any file with the same file id as `f`. Since `f` necessarily has only one file id (as defined in Section 2.1), then POP^f is the only possible minimal POP for `f` with respect to C (and `f` and `g` are in fact the same file).

To do this, we proceed by employing structural induction on their proofs of provenance, to prove that each POP slice must be equal.

Base case:

The first entries of POP^f and POP^g, called POP_0^f and POP_0^g respectively, must be identical.

File kernel:

Because `f` and `g` have the same file id `id`, and because each file's id is the hash of its file kernel (from the definition in Section 2.1), then `f` and `g` must have the same kernel.

Consequently `f` and `g` also must necessarily share the same *creator address* $\mathtt{a}_0$ and *issued cycle root* C_{j-1}, as these are both defined to be components of the file kernel.

It is worth noting that if the file id `id` doesn't have a (known) hash preimage of a suitable length to be a properly structured file kernel, then it cannot be a valid file, and has no POP. Similarly, if its issued cycle root C_{j-1} is not in C, then it is not a valid file in that sequence of cycles, and has no valid POP with respect to it. Finally, in the case of a null creator address; since any file is considered destroyed once it has the null address as an owner, f would be destroyed immediately upon creation, and does not ever exist as a valid file.

Cycle trie:

Because the issued cycle root is C_{j-1} for both `f` and `g`, the cycle root of POP_0^f and POP_0^g must be C_j as specified in Section 2.1.

Additionally since `f` and `g` share the creator address $\mathtt{a}_0$, they must share that as the address of the initial transaction (by the definition in Section 2.1). Further, by Section 2.5 that address $\mathtt{a}_0$ must be the key in the cycle trie for C_j that is associated with the txpx under which `f` and `g` are created. Consequently the corresponding cycle proof (for $\mathtt{a}_0$ in C_j) is common to POP_0^f and POP_0^g.

Further, the Merkle trie properties ensure there is at most one value $\mathtt{p}_0$ for key $\mathtt{a}_0$ in the cycle trie with root C_j, so that value must also be common to both POP_0^f and POP_0^g.

Note that if $\mathtt{p}_0$ is null, then Section 2.5 gives that $\mathtt{a}_0$ did not take any action in C_j. Because Section 2.2 requires $\mathtt{a}_0$ to act on a file whose file id is `id` in C_j for there to be a valid file with that file id, then neither `f` nor `g` exist as valid files.

It is also worth noting that $\mathtt{a}_0$ might not be an address corresponding to any known (private) keys (in fact the null address is taken to be a special case of this, in which the implicit consquences of not having a functional owner are made explicit). In this case, the cycle proof will most likely be null — though the case in which it isn't, is examined in the transaction packet.

Transaction packet:

Because Section 2.5 defines the value bound to $\mathtt{a}_0$ in C_j to be the hash of the txpx for that address in that cycle, then the assumed hash properties give either that there is a unique (known) hash preimage for p_0, or that it is impossible to construct POPs for `f` and `g`. This preimage being unique, it must be equivalent for POP_0^f and POP_0^g.

If the preimage is not the correct length to be a txpx, then $\mathtt{a}_0$ cannot have taken any valid actions in C_j, so neither `f` nor `g` exist as valid files, violating our precondition that `f` and `g` are files. The requirements of Section 2.4 also give that this is the case if the *current cycle root* in the txpx is not C_{j-1}, the address is not $\mathtt{a}_0$, or the *signature packet* does not constitute a valid signature of the first 96 bytes of the txpx.

It is worth noting that in the case of an address for which no known (private) keys exist, and for which the cycle proof in C_j is non-null, it is essentially impossible to have a valid signature packet. Thus the txpx (or cycle trie return value without a txpx as a preimage), provides that there is no valid creation event for either `f` or `g`, and neither file exists.

If however, the txpx is valid, then (from its structure in 2.4), it must contain a unique file trie root, F_0, which is therefore shared by POP_0^f and POP_0^g.

File trie:

Because the file trie root is F_0 for both POP_0^f and POP_0^g, they must have identical file tries by the assumed Merkle trie properties.

Because `f` and `g` share the file id `id`, Section 2.3 provides that the any operations taken by $\mathtt{a}_0$ in C_j on a file whose file id is `id`, must be represented in the value returned by the file trie, when `id` is used as a lookup key.

If this return value is null, then since both `f` and `g` have that file id, the

requirement that any operations on them be represented by that return value, implies that they cannot have been operated upon in C_j, and therefore cannot exist (by the issued cycle root requirement from Section 2.1).

Alternately, if there is a return value, the Merkle trie properties ensure it is a unique value $\mathtt{f}_0$ for the key `id` in the file trie with root F_0, and that value must be common to both POP_0^f and POP_0^g.

File detail:

Section 2.3 gives that $\mathtt{f}_0$ is the hash of the hash of the file detail describing any operation by $\mathtt{a}_0$, in C_j, on a file whose file id is `id`.

In the event that there is no file detail `d` such that $H(H(\mathtt{d})) = \mathtt{f}_0$, or `d` is not a valid file detail (e.g. it is not 96 bytes long, as required in Section 2.2), then the required creation operation cannot have happened, and once again neither `f` or `g` exist.

If the file detail `d` is a valid file detail, then the hash properties give that it must be the same for both POP_0^f and POP_0^g, because $\mathtt{f}_0$ is the same in each. Then `d` must contain the destination address $\mathtt{a}_1$ as its first 32 bytes, as defined in Section 2.2. In POP_0^f and POP_0^g the file detail `d` assigns `f` and `g`, respectively, to the destination address $\mathtt{a}_1$ at the end of C_j.

Moreover, the components of a POP slice are the file detail, the file proof, the transaction packet, the cycle proof, and the cycle root. Because each component of POP_0^f and POP_0^g has been shown to be equal, then these two POP slices must be equal, and we have completed the base case.

Inductive step: Show that POP^f and POP^g being equal through cycle C_{k-1} implies POP_k^f and POP_k^g must be equal as well.

The POP specification puts stringent requirements on the first slice of a file's POP. If those requirements are not met then it is not a TODA file: it was not created, so it never existed.

Once a file has been created, it may also be destroyed. This could happen by e.g. sending a file to the null address, which burns the file. Attempting to add POP entries subsequent to the file being destroyed invalidates the POP.

As such, if `f` was destroyed prior to C_k then POP_k^f and POP_k^g must be equally non-existent. We will then limit our consideration to cases where `f` is still viable, but must also consider that `f` may be destroyed in cycle C_k.

Cycle trie:

As in the base case, the Merkle trie properties guarantee that there is a unique cycle trie whose root is C_k.

Let $\mathtt{a}_k$ be the address which owns `f` and `g` at the end of C_{k-1} By definition of ownership, this owner would be the address in the file detail of POP_{k-i}^f and POP_{k-i}^g, for the lowest value of $i \geq 1$ for which POP_{k-i}^f and POP_{k-i}^g (being equal by our induction assumption, because $i \geq 1$) contain a file detail. Since the base case necessarily has a file detail, such an i must exist.

This ownership imposes the requirement (from the definition of a POP) that $\mathtt{a}_k$ must be the key for the cycle trie in POP_k^f and POP_k^g. The Merkle trie properties ensure that there is at most one value $\mathtt{p}_k$ for key $\mathtt{a}_k$ in the cycle trie with root C_k, so that value (possibly null) and its cycle proof must be common to both POP_k^f and POP_k^g.

If $\mathtt{p}_k$ is null, that indicates that address $\mathtt{a}_k$ did not operate on any files in C_k and continues to own `f` and `g` at the end of the cycle. This cycle proof also constitutes the entire POP slice for each of POP_k^f and POP_k^g, satisfying the induction conclusion.

In the cases where $\mathtt{p}_k$ is not a valid hash output (i.e. it is the wrong length), there is no valid POP for either `f` or `g` past C_{k-1}. This case being contrary to the assumptions in the theorem, can be ignored.

In the remaining cases, we proceed to inspect the transaction packet that hashes to $\mathtt{p}_k$

Transaction packet:

It may be the case that the hash preimage of $\mathtt{p}_k$ is not known, in which case it is impossible to construct a conclusive POP slice for `f` and `g` in C_k, (as the txpx that is the preimage of $\mathtt{p}_k$ is a necessary component of the POP slice when $\mathtt{p}_k$ is not null), and is therefore impossible to construct a POP for either file. This satisfies the conclusion of the theorem that there is *at most* one valid POP for a given file and cycle sequence.

If however, the expected holds and $\mathtt{p}_k$ has a hash preimage, then this preimage is treated as unique (in accordance with the hash properties), and expected to be a transaction packet satisfying certain properties, namely:

- that its *current cycle root* is C_{k-1}
- that its address is $\mathtt{a}_k$
- the signature represented by the signature hash is a valid signature of the packet's *current cycle root*, *file trie root*, and *expansion slot* (concatenated in that order).

If any of these properties fail to hold, then uniqueness of the hash preimage of $\mathtt{p}_k$ gives there cannot be a POP for either `f` or `g` past C_{k-1}. This case contradicts the assumptions of the theorem, and can be ignored.

In the (expected) case that all these properties hold, the txpx is unique, and it contains a single file trie root F_k (by its definition in Section 2.4). These are therefore shared by POP_k^f and POP_k^g.

File trie:

Because the file trie root is F_k for both POP_k^f and POP_k^g, they must have identical file tries by the Merkle trie properties.

Because `f` and `g` share the file id `id`, the definition of a file proof in Section 2.3 provides that this must be the key used in the file proof of POP_k^f and POP_k^g.

The Merkle trie properties ensure that there is at most one value $\mathtt{f}_k$ for key `id` in the file trie with root F_k, so that value must be common to both POP_k^f and POP_k^g.

That value might be null, indicating that address $\mathtt{a}_k$ did nothing with `f` in that cycle. By the definition of a POP slice, when there is a null value, its file proof completes the POP slice, and provides the conclusion of the induction step.

File detail:

The definition of the file trie in Section 2.3 provides that the value associated with each key by the file trie is either null, or the hash of a hash of a file detail. Having already considered the null case, we now examine the case where $\mathtt{f}_k$ is not null.

If $\mathtt{f}_k$ is the wrong length to be a hash, then it cannot possibly be the hash of a hash of a file detail; this is contrary to our theorem precondition that `f` and `g` have valid proofs, and a case that we can ignore.

It may also be the case that there is no (known) `d` such that $H(H(\mathtt{d})) = \mathtt{f}_0$. In this case, a valid proof cannot be constructed, and like in the other cases of missing hash preimages, we satisfy the requirement that there is *at most* one valid POP.

The remaining cases now provide that there is a `d` such that $H(H(\mathtt{d})) = \mathtt{f}_0$. The first such case to examine is that `d` cannot be a valid file detail (e.g. it is not 96 bytes long, as required in Section 2.2). This constitutes a contradiction of our assumption that `f` and `g` have valid POPs, and can be ignored.

Then `d` must be a valid file detail, and its first 32 bytes are the destination address $\mathtt{a}_{k+1}$ for both `f` and `g`. This completes both of their proofs identically, providing the same owner $\mathtt{a}_{k+1}$ at the end of C_k, and satisfying the requirements of the induction step.

Having exhausted all possible cases for the induction step, and satisfied its requirement in each case, and having proven the desired result for the base case, we have by induction that `f` and `g` have the same minimal POP. Thus, for any file `f` with id `id` and a sequence C of cycle roots $\{C_0, C_1, ..., C_n\}$ which contains `f`'s cycle of creation, then `f` has a single minimal POP with respect to C. □

3.1 Unique Ownership of Each File

Corollary 1. *There is at most one address that can prove ownership of a file `f` in cycle k.*

Proof. From the proof of uniqueness of the POP, which depends on unique ownership of `f` at the end of each cycle in a (partial) POP, and from a POP being the exclusive mechanism to prove ownership of a file, the desired result follows automatically. □