Introduction

Chainscript is an open JSON Standard for Proof of Process.

Proof of Process is a scalable protocol that allows multiple partners to trust a common process, or a workflow, by decoupling the proof of data from the secret data in a way that results in a single contextual proof that spans all the steps of a process.

Read the Whitepaper

An Example

Imagine a digital process to notarize a document.

With Chainscript, each time you notarize a document, you would create a **Chain Map** which records all the notarization steps.

Each step here is referred to as a **Link** in the Chain Map.

Examples of Links would be:

- Uploading the Document
- Emailing requests to sign to notarizers
- Receiving the signatures
- Confirming the signatures
- Publishing a final report once enough notarizers have signed

You can record the above process in Chainscript where each step will be recorded as a Link in a Chain Map. Once recorded, the Links are permanent and unchangable.

Useful Features

Some of the things that you can do with such an Chain Map are:

- **Automation & Playback**: Set the order and sequence of the steps for complete automation and playback
- **Tracking**: "Color" specific types of steps in a process so you can track it and index it
- **Concurrent Steps**: Include concurrent steps as the Chain Map branches out into a multidimensional graph where concurrent steps share the same "color"

- **Notarized (trusted) steps**: The existence of each step in a process can be objectively proved with a timestamp

Links and States

Consider the example of the above process of notarizing a document. Let's take the first step of uploading a file. Here we will store the name of the file uploaded and the user who uploaded it in JSON. We will refer to this user defined data for a specific step as a `state`.

```
"state": {
  fileName: "example.pdf",
  fileHash: "4ee15684e24350ff88a8d9ef7761d506faad7bf3f793c2ea5d89b0fbe906603f",
  uploadedBy: "John Smith"
}
```

Now lets introduce some context to the above `state`. We will do this by adding metadata or `meta` to the state. For example, we can add a title, tags, a priority index, and the time when the step was executed.

```
"meta": {
  "title" : "File Info for Notarizing",
  "tags" : ["notarize", "filename", "more_list", "of", "tags"],
  "priority": "0"
  "updatedAt": 1455532426303
}
```

A `state` and its `meta` is wrapped in what we call a `link`.

A `link` *is immutable*, i.e., once created, it can not be changed. New links can be added, existing ones can not be changed.

Besides the above user defined fields in the above `meta`, there are others too which need to populated inside each `meta` by the platform implementing Chainscript. We will discuss these in a little while.

Segments and Chain Maps

Segments

A **Segment** represents a single step in a process.

A `segment` is simply a `link` with its `meta`.

A **Root Segment** sets the initial conditions for the process.

Chain Maps

A process is represented by one or multiple hash chains of Segments.

In this context a process is referred to as a **Chain Map**, as it maps out the process in the form of one or multiple hash chains.

Each step in a process is represented by a single `segment`. Many such segments make up a process, or what we refer to as a `chain map` or simply, `map`.

A `chain map` can have many branches. Each unique path in a `chain map` is referred to as a `chain`.

All Together

The metadata of a State has a field for the hash of the previous Link to bind each Segment with its predecessor. The metadata of the State also has fields for both the hash of the State and the unique ID for the Chain Map to which the Segment belongs.

The metadata of the Segment also contains the hash of the current Link and any Vendor specific information that can be stored in the `x<ServiceProvider>`which is `xStratumn` in the example below.

```
"segment": {
  "link": {
    "state": {
      fileName: "example.pdf";
      uploadedBy: "John Smith";
    },
    "meta": {
      "title" : "File Info for Notarizing",
      "tags" : ["notarize", "filename", "more_list", "of", "tags"],
      "priority": "0"
      "updatedAt": 1455532426303,
      "mapId": "56c11d0ea55773bc6e681fde",
      "agentHash":
"f559625364c117e9bf07f9aa536cf72b1f1468e483b16f0b45e10e13faf8c8d9",
      "stateHash":
"06faad7bf3f793c2ea5d89b0fbe906603f4ee15684e24350ff88a8d9ef7761d5",
      "prevLinkHash":
"9b03d05c07fb44fe5aa472115cd918e886b71e0dbc6fe06ef976566c98272ee9"
    }
  },
  "meta": {
    "linkHash": "fa871e81c469fa947eacd40f89dc5627a0cb3a96551a651c034787c752d4448e",
```

```
    "evidence": {
      ...
    },
    xStratumn: {
      totalTimeMs: 16.941432,
      loadApplicationCache: "miss",
      loadApplicationTimeMs: 5.387555,
      loadInputLinkCache: "miss",
      loadInputLinkTimeMs: 10.388065
    }
  }
}
```

Scripts and Agents

Scripts

Scripts are the logic responsible for executing one or multiple **Transitions** from one Segment to the next. In computer science terms, it means defining functions to transition between two states:

T(state, ...args) => state'

Agents

Agents are private computing units responsible for executing Scripts.

Here is an example of a Script with two Transitions that are each represented by a unique function:

> **Vendor Specificity**
>
> The calls mentioned here, `this.append()` and `this.reject()`, are vendor specific; these particular examples are used by Stratumn.

```
module.exports = {

  /** Transition function to initialize the process */
  /** Creates the first segment with the file and the user info in it */

  init: function(fileToNotarize, uploadedBy) {
    this.state.fileToNotarize = fileToNotarize;
    this.state.uploadedBy = uploadedBy;
```

```
    this.append();
    return;
  },

  /** Transition function to send emails asking for signatures */
  /**Creates the segment following the first one */

  emailToAskForSigns: function(listOfUsersToEmail, fromEmail, titleEmail, bodyEmail){
    var email = new Email();
    email.addToList(listofUsersToEmail);
    email.setFrom(fromEmail);
    email.setSubject(titleEmail);
    email.setHtml(bodyEmail);

    this.state.emailContent = email;

    email.send(function(response) {
      this.state.emailResponse = response;
      this.append();
      return;
    });
  }
}
```

The function `Agent#init()` is responsible for creating the Root Segment in this example.

When entering the function, `this.state` will be an empty object. You must call either:

- `this.append()` to create the first link
- `this.reject()` if an error occurred (for instance if the arguments are invalid)

The other functions can be named however you want. The only difference from `Agent#init()` is that `this.state` will initially be set to the state of the current Link.

Because you are resolving by calling functions instead of returning values, it is possible to call asynchronous functions.

However make sure to call `this.append()` or `this.reject()` only once.

State and Transition Metadata

Every time a new Segment is added (a Transition occurs), the platform implementing Chainscript needs to store the following information as well:

- The ID for the Chain Map to which the `segment` belongs
- The hash of the Agent file
- The hash of the State of the Segment
- The hash of the previous Segment's Link
- The name of the Transition that was executed to create and validate this Segment
- The list of arguments passed to the Transition

Thus, the complete structure of the Segment would be:

> **Vendor Specificity**
>
> Stratumn stores Transition metadata under `action`.

```
"segment": {
  "link": {
    "state": {
      fileName: "example.pdf";
      uploadedBy: "John Smith";
    },
    "meta": {
      "title" : "File Info for Notarizing",
      "tags" : ["notarize", "filename", "more_list", "of", "tags"],
      "priority": "0"
      "updatedAt": 1455532426303,
      "mapId": "56c11d0ea55773bc6e681fde",
      "agentHash":
"f559625364c117e9bf07f9aa536cf72b1f1468e483b16f0b45e10e13faf8c8d9",
      "stateHash":
"06faad7bf3f793c2ea5d89b0fbe906603f4ee15684e24350ff88a8d9ef7761d5",
      "prevLinkHash":
"9b03d05c07fb44fe5aa472115cd918e886b71e0dbc6fe06ef976566c98272ee9",
      "action": "init",
      "args": ["example.PDF", "John Smith"]
    }
  },
  "meta": {
    "linkHash": "fa871e81c469fa947eacd40f89dc5627a0cb3a96551a651c034787c752d4448e",
    "evidence": {
      ...
    },
    xStratumn: {
      totalTimeMs: 16.941432,
      loadApplicationCache: "miss",
      loadApplicationTimeMs: 5.387555,
      loadInputLinkCache: "miss",
      loadInputLinkTimeMs: 10.388065
    }
  }
}
```

Proof of Existence

Segments are notarized via the blockchain in fixed time intervals. In above example, this is backed by a **merkle proof** and a blockchain transaction with the merkle proof in it, which in the case of Bitcoin Blockchain would be using OP_RETURN.

The merkle proof can be found in the `meta.evidence`.

In the above Segment snippet, the `meta.state` is “COMPLETE”, which indicates that this transaction has been broadcasted to the blockchain network. This allows others to validate the audit trail of the Segment using methods that are publicly accessible.

Merkle Trees

A merkle tree is a data structure that is able to take *n* number of hashes and represent it with a single hash.

Merkle tree proof is generated by taking the hash of one leaf of the tree and hashing it with its neighbouring hash. The parent hash then hashes with its neighbour until the root hash is calculated. Leaf hashes can be verified against the root hash by following the tree from leaf and neighbour back to the root.

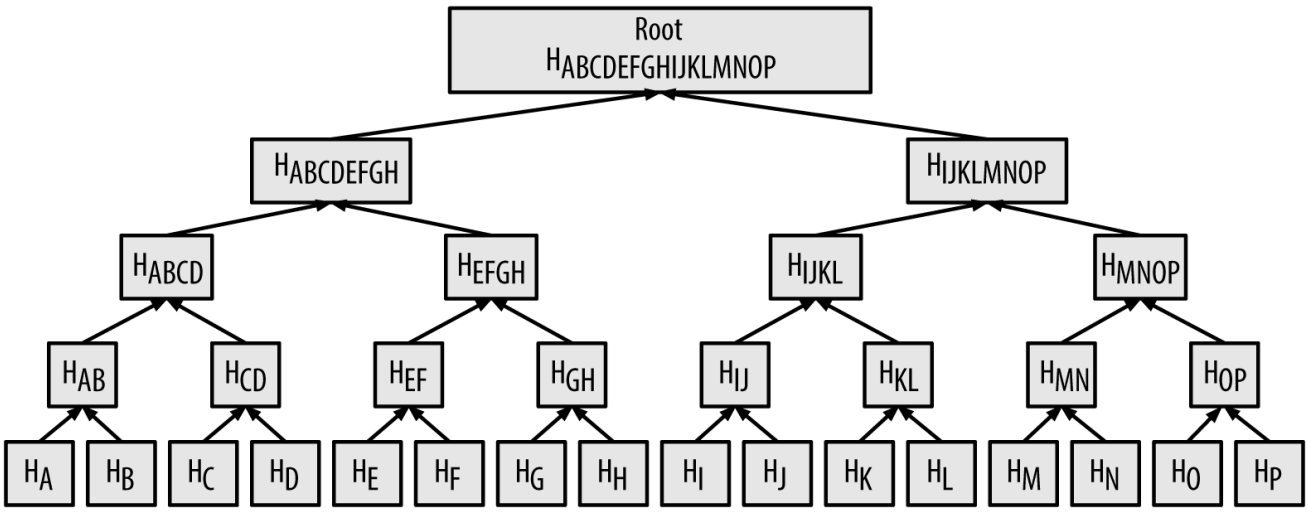

Image via Wikipedia. Learn More about Merkle Trees on Wikipedia.

The Merkle root represents all the hashes in the tree where if one hash is modified the entire tree is invalid. **Thus, with the Merkle tree path corresponding with the Blockchain transaction, one can guarantee the proof-of-existence of the unmodified dataset**.

From the Merkle Proof the path can be validated by concatenating the left and right hash, and validating the result against the parent. This step is repeated until the merkle root is found.

The transaction hash and blockchain is included in the Objective Evidence for validating the Merkle Root. As an example, for the Bitcoin Blockchain, the merkle root is stored using the OP_RETURN opt code.

Hash Generation

Because Chainscript is based on JSON, we have to create a canonical string from the JSON before hashing it. To do so, we can use the canonical-json node module.

After the canonical string is generated, we use the SHA256 algorithm to generate the hash.

Here's an example to do it using Node.js:

```
var crypto = require('crypto');

// The `canonical-json` package is required.
// You can install it with `npm install canonical-json`.

var stringify = require('canonical-json');

// Computes the canonical hash of a JSON object.

function hashJson(obj) {
var hash = crypto.createHash('sha256');
hash.update(stringify(obj));
return hash.digest();
}
```

Merkle Proofs

You can verify that a Merkle Proof is valid on your own without using any proprietary tools. It uses standard cryptographic functions which are available in many programming languages.

Here is a script to do it using Node.js:

```
var crypto = require('crypto');

// The `canonical-json` package is required.
// You can install it with `npm install canonical-json`.
var stringify = require('canonical-json');

// Computes the canonical hash of a JSON object.
function hashJson(obj) {
var hash = crypto.createHash('sha256');
hash.update(stringify(obj));
return hash.digest();
}

// Returns whether the merkle proof of a Chainscript state is valid.
function verifyMerkleProof(segment) {
var hash = hashJson(segment.link);
var evidence = segment.meta.evidence;

var merklePath = evidence.merklePath;

// Check levels one by one
for (var i = 0; i < merklePath.length; i++) {
  var level = merklePath[i];
  var left = new Buffer(level.left, 'hex');

  if (level.right) {
```

```
    var right = new Buffer(level.right, 'hex');
  }

  // Make sure hash is in the current level
  if (hash.compare(left) !== 0 && (!right || hash.compare(right) !== 0)) {
    return false;
  }

  // Compute parent hash
  if (right) {
    var parent = crypto
      .createHash('sha256')
      .update(Buffer.concat([left, right]))
      .digest();
  } else {
    parent = left;
  }

  // Make sure parent hash is corrent
  if (parent.compare(new Buffer(level.parent, 'hex')) !== 0) {
    return false;
  }

  hash = parent;
}

// Make sure root is correct
return hash.compare(new Buffer(evidence.merkleRoot, 'hex')) === 0;
}

module.exports = verifyMerkleProof;
```

Template Chainscript

JSON Structure

```
"chain map": [              // Each ChainMap consists of a set of segments in a
certain order
"segment": {              // Wrapper
  "link": {               // - I M M U T A B L E -
    "state": {            // User defined data
      .....
      .....
      .....
    },
    "meta": {             // Link Meta
      "title" : "",       // User defined: title for the state
      "tags" : [],        // User defined: to "color" chains
      "priority": ""      // User defined: sorting key
      "updatedAt": ,      // User defined: timestamp of when it was created
      "chainId": "",      // Unique ID of the ChainMap
```

```
      "agentHash": "",    // Hash of the agent
      "stateHash": "",    // Hash of the state
      "prevLinkHash": "", // Hash of the prev link
      "action": "",       // Name of the transition
      "args": [],         // Arguments passed to the transition
    }
  },
  meta: {                 // Segment Meta : - M U T A B L E -
    linkHash: "",         // Hash of the current link
    evidence: {           // Objective evidence: Blockchain or Consensus Info
      state: "",          // QUEUED or COMPLETE : state of Blockchain entry
      "merkleRoot": "",   // Is populated once the state above is Compete
      "merklePath": [],   // Is populated once the state above is Compete
      "transactions": {   // Transaction ID from the Blockchain or Consensus System
        "[blockchain]:[network]": ""
      }
    },
    "x<ServiceProvider>": { // Vendor Specific Info
      ....
    }
  }
}, ...
]
```

Sample Chainscript

```
"segment": {
{
  "link": {
    "state": {
      "fileToNotarize": "test.PDF"
    },
    "meta": {
      "title" : "File Info for Notarizing",
      "tags" : ["notarize", "filename", "more_list", "of", "tags"],
      "priority": "0"
      "updatedAt": 1455532426303,
      "chainId": "56c11d0ea55773bc6e681fde",
      "agentHash":
"f559625364c117e9bf07f9aa536cf72b1f1468e483b16f0b45e10e13faf8c8d9",
      "stateHash":
"06faad7bf3f793c2ea5d89b0fbe906603f4ee15684e24350ff88a8d9ef7761d5",
      "prevLinkHash":
"9b03d05c07fb44fe5aa472115cd918e886b71e0dbc6fe06ef976566c98272ee9",
      "action": "addFileToNotarize",
      "args": ["test.PDF"],

    }
  },
  "meta": {
    "linkHash": "fa871e81c469fa947eacd40f89dc5627a0cb3a96551a651c034787c752d4448e",
    "evidence": {
      "state": "COMPLETE",
      "merkleRoot":
"855e019f67bc5ba09b7efb3a9e169bbe14e6abafa5b02d686607e25107db1b4c",
      "merklePath": [
```

```
        {
          left: "fa871e81c469fa947eacd40f89dc5627a0cb3a96551a651c034787c752d4448e",
          right: "aa1331b6f1155fff5865116e5adcb6e4cdf0435fc2bd68053a4d24795f47dad3",
          parent: "39b43545134a903dd16be3d0b5391474ef82241acd33e8496b103ee8d378efea"
        },
        {
          left: "39b43545134a903dd16be3d0b5391474ef82241acd33e8496b103ee8d378efea",
          right: "9fb3164966231fca20c75a562d0b4c1e7adc68738ac6c179dd1ecf921845dbfd",
          parent: "fedacd347e84d39ab3122f78d762dbbc79646704deb90fbb118b140c85f4df96"
        },
        {
          left: "d7fa0ed22e4b58eb227a826a1d2f0ef564f5052172bd36fe612a2fa214d452c8",
          right: "fedacd347e84d39ab3122f78d762dbbc79646704deb90fbb118b140c85f4df96",
          parent: "7e7dec576382e948f4980e4295f6ef9b89748e4586723a7b9481ee20cfb14db5"
        },
        {
          left: "2c0ada2bdea45977acd0a4d56fda6f970f34df32a1e507e99cd7284c0038b22a",
          right: "7e7dec576382e948f4980e4295f6ef9b89748e4586723a7b9481ee20cfb14db5",
          parent: "fe7214757ed9d701a2ed112d01c1b17a734094444ee067dd2ee4c14907419a24"
        },
        {
          left: "dae80a5aa2b397285ce6ca079c78dad737099d293a8783d4b8680efff57e921e",
          right: "fe7214757ed9d701a2ed112d01c1b17a734094444ee067dd2ee4c14907419a24",
          parent: "5f9343488c91f22a0cfea5c9797e244c1a6d20872048e157e515a1172678ebcf"
        }
      ],
      "transactions": {
        "bitcoin:testnet":
"15359dc0ef4c430d6219d07e0dd7be96442af20ffd6f79727559028a06518474"
      }
    },
    "xStratumn": {
      "totalTimeMs": 16.941432,
      "loadApplicationCache": "miss",
      "loadApplicationTimeMs": 5.387555,
      "loadInputLinkCache": "miss",
      "loadInputLinkTimeMs": 10.388065
    }
  }
}
}
```


- `link` is the actual link
  - `state` is the user defined state which captures a specific step
  - `meta` is the meta data of the link
    - `link.meta.title` is a user defined title for the state
    - `link.meta.tags` is used to "color" chains so that they can be indexed in a multidimensional graph
    - `link.meta.priority` is used to "order" links in a chain so that they can be sorted in a multidimensional graph
    - `link.meta.mapId` is the chain ID common to all the links in the chain
    - `link.meta.agentHash` is the hash of the agent
    - `link.meta.stateHash` is the hash of the state
    - `link.meta.prevLinkHash` is the hash of the previous link, if any
    - `link.meta.action` is the name of the method that was called to append this link

        - `link.meta.args` is the list of arguments that were passed to the method, if any
- `meta` is the meta data of the segment
    - `linkHash` is the hash of the link
    - `evidence` is information that shows the link hash was inserted in one or multiple blockchains
        - `state` is the state of the proof, either `QUEUED` or `COMPLETE`
        - `merkleRoot` is the root of the Merkle tree which is inserted in one or multiple blockchains
        - `merklePath` is the path to go from the link hash to the root in the Merkle tree
            - `merklePath[].left` is the left hash of the Merkle node
            - `merklePath[].right` is the right hash of the Merkle node
            - `merklePath[].parent` is the parent hash of the Merkle node
        - `transactions` is the transactions that were created in one or multiple blockchains
            - `[blockchain]:[network]` is a transaction ID or hash, in this example it is `[bitcoin]:[testnet]`

For the sample Chainscript above, click here to see the actual Bitcoin Testnet Transaction.

Glossary

(Chain) Map
Hash Chain of Segments. Represents a process.

Hash Chain
Ordered set of elements with hashed keys such that each element contains the hash of its previous element.

Link
State and metadata about the state. Immutable.

Link metadata
Contains Proof of Existence. Mutable.

Segment
Link and metadata about the Link. Represents a single step in a process.

State
User defined data to capture a specific step in a process.