Skip Ledger: a Commitment Scheme for Ledgers

DRAFT

Babak Farhang
babak@crums.io

January 2025

Abstract

This paper introduces Skip Ledger, a novel commitment scheme designed for evolving, append-only lists, or simply ledgers. Unlike traditional hash chains, skip ledger embeds a binary hash tree structure in the calculation of each row's hash, enabling logarithmically succinct proofs of inclusion and order for any set of ledger rows. The hash structure enabling efficient partial reveals of ledger contents is described, then formalized in pseudo-code by the defining a function that calculates a row's commitment hash, in terms of both the row number and a user-defined function that returns the hash of the *contents* of a ledger row. Besides enabling succinct proofs of ledger row contents, the scheme's perhaps most notable property is that it also enables succinct proofs of the order of the commitments, themselves. The scheme's efficiency in areas requiring audit, integrity, and non-repudiation are discussed and contrasted with other approaches. The scheme's applicability to blockchain protocols (more efficient off-chain proofs) is noted. Finally, the paper posits if blockchains can create tokens of value in their ledger entries, then so too can private ledgers.

1. Introduction

Ledgers are historical lists of things. Whether by markings on a stick, carvings in clay, or ink on paper, ledgers have always recorded human activity. And while the technologies for maintaining the veracity of those records have evolved, one principle has remained constant: the technology should make tampering with the records difficult (dry clay is not editable, for example), so that tampered records are readily apparent. This, then, is another modern take on those markings on a stick.

Traditional methods like hash chains [HS91] provide strong security guarantees about order (which record precedes another, for example), but suffer from inefficiencies when it comes to *verifying* that order. In particular, with most hash chains, verifying order requires reading the chain from the beginning. Since hash chains are typically large, it's difficult to package succinct off-chain proofs of order.

Skip ledger, however, enables succinct, off-chain constructions for proof of order. A sketch of the typical usage model under the scheme goes something like this:

> Given the commitment hash H_N for block N, establish that it encodes the contents (or commitment hash H_n) of block n in a succinct proof (where $0 < n \leq N$) .

That is, if a user has reason to believe they know (or can trust) the commitment hash H_N for block N, then they can be convinced about the contents of any block at or before N using a compact, off-chain hash proof.

After a brief review of prior related work, the scheme's data model and commitment scheme are defined in terms of a user-defined `input_hash(n)` function that returns the hash of the contents of the ledger row numbered *n*, and the [fixed] commitment hash function, `row_hash(n)`. Next, the scheme's properties are enumerated (what qualifies it as a commitment scheme; what makes skip ledger a useful primitive) with emphasis on the scheme's fundamental proof structure called a *path*. Paths are to skip ledgers what Merkle proofs are to Merkle trees; but unlike Merkle proofs which model *fixed*, not append-only, collections, paths are composable from (and *de*-composable to) sub-paths. An informal "mini-algebra" about the row numbers in the composition of paths is introduced to justify some of the claimed properties. The *path* proof structure is only described in words, referencing only the already defined `input_hash` and `row_hash` functions; pseudo code for the actual packaging of proofs (paths) is not included.

The `row_hash` function, as defined recursively in the pseudo code, is inefficient. The user-defined, ledger-specific `input_hash` function too may be inefficient. The section on skip ledger chains outlines a simple "memo-ization" technique that allows a ledger owner to generate paths (succinct proofs of order and inclusion) in constant time; other use cases that involve less frequent recordings of a ledger's commitment hash (where there are no immediate plans for revealing anything), use a succinct memo-ization structure (called *hash frontier*) to efficiently calculate row (commitment) hashes forward, in a "single-pass".

The Discussion section compares and contrasts the scheme with other approaches, emphasizing its applicability in a range of use cases. Ledgers (append-only collections), after all, model many things: log files, streams, sequences of "data frames", journals, private ledgers, public ledgers, blockchains, or other kinds of objects yet imagined. The scheme's general applicability and efficiency in the areas of audit, proof of provenance, non-repudiation (etc.) are analyzed. The section closes with a hypothetical warehouse ledger, using it to suggest that private ledgers (backed by business operations) can encode and expose tokens of value much like blockchains do.

1.1 Prior / Related Work

Both Merkle trees [Mer78][Mer82] and hash chains, while prominently featured in blockchain technology [Nak08], have a rich history. These primitives are used in various settings, including

generating one-time keys in Lamport signatures [Lam79][Lam81], and providing verifiable time stamps for digital documents [HS91].

Hash proof structures often mirror analogous, conventional data structures. Merkle trees, for example, mirror binary trees; hash chains are similar to linked lists. The word *mirror* here also conveys the idea that the references in these hash proof structures point in the *opposite direction* of their conventional counterparts. For example, the hash references (commonly called *hash pointers*) in a Merkle proof are constructed from the bottom up , from leaf to root, rather than vice versa (as in a conventional binary tree). Skip ledger also mirrors a conventional type of data structure: the skip list [Pug90].

The use of skip list-like hash structures for cryptographic linking (and commitment) is not a new concept. For example, the NIST Randomness Beacon [KBPB19] records (commits) successive beacon hashes in a skip list-like hash structure organized like the Gregorian calendar (year/month/day, etc.). Although skip lists are mentioned or used in prior, related work [Mey23], skip ledger offers a more concrete skip-list-based commitment scheme that is applicable in more general settings.

2. Model

A ledger is modeled as an append-only list of rows (items), each row represented by a hash value derived from the row's contents (a line of text, a row in a database, a "chunk" of a stream, etc.) using a cryptographically strong hash function ***H***. In skip ledger terminology, this hash value is called the row's *input-hash*. Additionally, each skip ledger row has a computable *commitment hash*, which in skip ledger terminology is just called the *row-hash*. That is, a row-hash at row n also stands in as a commitment hash for the ledger when it had n rows.

Indeed this is an already familiar concept in blockchain [Nak08]: the hash of the n-th block is a commitment for the state of the entire chain when it had n blocks. There, each block is linked to its predecessor using a special hash cell that simply records the hash of the previous block. Under the skip ledger, each row's (block's) hash is computed as the root of a binary hash tree over that and all previous rows in the ledger. This property, in turn, allows one to efficiently reveal and prove the contents of any row against *any* commitment hash of the ledger after it had that row (block).

2.1 Row Hash

Rows are numbered, starting from 1, monotonically increasing, with no gaps. Every computed row hash encodes the hash of the preceding row, but also, depending on the row no. n, k-many more hashes of predecessor rows, where k is the number of times 2 divides n:

$k = \max \{ i \in \mathbb{N}_0 \text{ s.t. } 2^i \mid n \}$

Each row's hash "directly" encodes $k + 1$ predecessor row hashes. The hashes referenced from Row [n] are those of rows numbered

$$n - 2^0, n - 2^1, n - 2^2, .. , n - 2^k$$

or more simply

$$n - 1, n - 2, n - 4, .. , n - 2^k.$$

The hash of row [n] is computed by hashing the concatenation of the row's *input hash* (computed from the source row) together with the root hash of a Merkle tree constructed from the row hashes of the predecessor rows numbered above.

The following pseudo code fleshes out a recursive definition for the row hash:

```
// Returns the "hash" of the byte sequence m using a strong cryptographic
// hashing algorithm (e.g. SHA-256). The hash of the empty [byte] sequence
// H("") is defined as a string of zero bits with the same length as the
// usual output. This sentinel value is denoted H_o below.
//
function H(m) :


// Computes and returns the Merkle root hash over the given ordered list of hashes
// using a strong cryptographic hash function. The algorithm here does not prepend
// internal and leaf hash nodes (with 0 and 1): since the no. of hashes is predetermined
// in this application, the so-called "2nd-preimage attack" does not apply.
//
function merkle_root(hashes) :
    if (length(hashes) == 0) return H_o            // (never reached in this pseudo code)
    if (length(hashes) == 1) return hashes[0]      // hash of singleton is self

    while (length(hashes) > 1) {
       new_hashes ← []
       for i ← 0 to length(hashes) - 1 step 2 {
           if (i + 1 < length(hashes))
              new_hashes.push( H(hashes[i] + hashes[i+1]) )
           else
              new_hashes.push( H(hashes[i]) )
           }
       hashes ← new_hashes
    }
    return hashes[0]


// Computes and returns the hash of the source row in the ledger
// numbered n (> 0). This may the "straight" hash of a byte sequence, or
// a "composition" of the hashes of cell values in row [n].
//
function input_hash(n) :
    // .. compute hash of ledger source row (or read memo-ized result)
    return h
```

```
// Returns the commitment hash for the ledger row numbered n.
//
function row_hash(n) :
    // check for the sentinel row [0]
    if (n == 0)
        return H_o                  // (zeroed hash)

    // below, '+' means concatenate
    return H( input_hash(n) + link_merkle_hash(n) )


// Returns the Merkle root hash for the ledger row numbered n.
//
function link_merkle_hash(n) :
    pn ← pointer_nos(n)
    ptr_count = length(pn)
    prev_row_hashes ← [ptr_count]   // an array of hashes (or other container) with
                                    // ptr_count-many slots
    for i ← 0 to ptr_count - 1 do

        prev_row_hashes[i] ← row_hash(pn[i])

    return merkle_root(prev_row_hashes)


// Returns the row no.s row n's hash is derived from. These are deterministically
// set by row no. (unlike the randomization in the skip list search structure).
//
function pointer_nos(n) :
    count ← skip_count(n)
    PN ← [count]      // an array (or other container) with count-many slots

    for i ← 0 to count - 1 do
        PN[i] ← n - 2^i

    return PN


// Returns the no. of skip [hash] pointers to previous rows for the row numbered n.
// I.e. returns 1 +  max { i ∈ ℕ₀ s.t. 2^i | n }
//
function skip_count(n) :
    e ← trailing_zero_bits(n)// no. of trailing zero bits in big-endian representation
    return e + 1
```

Note, in practice a `row_hash` implementation will involve some form of memo-ization (in memory, or on disk); otherwise, each access of `row_hash(`*n*`)` will cost **O**(*n*) operations.

3. Properties

The `row_hash` function above has sufficient properties to satisfy as a *commitment scheme*.

3.1 Commitment

The hash of any row numbered *n* (see `row_hash`) acts as a commitment to the contents of a ledger with *n* rows.

3.2 Binding

Once the hash of row *n* is published, it is computationally difficult to change the contents of any row numbered *n* or lower without changing the hash of row *n*. This follows from the fact that the hash of every row always depends on the hash of the immediate row before it.

3.3 Hiding

The hash of a row numbered *n* does not reveal anything about the contents of the ledger with *n* rows, even if *n* (the row no.) is revealed.

3.4 Partial Reveal

The commitment scheme enables 2 layers of differential information disclosure: general and specific.

3.4.1 Path (General)

Once the hash of row *N* is published (committed to), the ledger owner may later reveal how the hash row *N* is derived from the hash of any row numbered $n < N$. In skip ledger terminology, such proofs are called *paths*. Since hashes don't leak information, the only information conveyed is about the row numbers themselves.

Definition 1. *Full row.*

A unit of commitment information about a ledger row. It is composed of 3 parts:

1. the row number *n*.
2. the row's *input-hash* (`input_hash(n)`).
3. the row's *link-Merkle-hash* (see `link_merkle_hash(n)`).

The row's (commitment) hash is computable from items (2) and (3).

Definition 2. *Path.*

A sequence of strictly ascending (numbered) *full rows* along with *link* Merkle *proofs* establishing each successive full row's *link-Merkle-hash* is derived from the row hash of the previous full row.

Definition 3. *Skip path.*

The shortest path linking two numbered rows (the one with the fewest *full* rows) is called the *skip path* (for those numbered rows).

```
// Returns the minimum-length ascending list of full row numbers in a path.
// beginning with row number a and ending with row number b (a < b).
//
function skip_path_nos(a, b) :

    // assemble the no.s in descending order
    nums ← []
    last ← b
    nums.push(last)

    while (last > a) do

        for i ← skip_count(last) - 1 to 0 step -1 do
            next ← last - 2^i
            if (next >= a) {
                last ← next
                nums.push(last)
                break    // ..from for loop
            }

    // return in reversed (ascending) order
    return reverse(nums)
```

The next section explains the mechanics of paths using a particular example.

3.4.2 State (General)

Once the hash of row [*n*] is published, one may later reveal (and prove) this hash belongs to a ledger with *n* rows. This is achieved using a path (the hash proof) linking the hash of row [*n*] to the hash of the first row [*1*]. For example, if *n* is 534, then the path contains information about constructing the hashes of the following row no.s:

[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 528, 532, 534]

These are the path's *full* row numbers. From left to right, the hash of each row is determined by

1. The input hash (the hash of the source data) at that row no.
2. A Merkle proof linking the hash of the row no. on the left to Merkle root hash. (The first row [1] links to the sentinel row [0] whose hash is a string of zeroed bits.)
3. The full row's (commitment) hash is the hash of the concatenation of the row's input hash (1) and the (link) Merkle root hash (2).

Since row [0], the sentinel row, is only referenced "directly" (via a row's link Merkle tree) by rows numbered 2^k (k = 1, 2, 3, ..), the structure of the hash proof reveals n, the number of rows in the ledger with the given commitment hash.

Note the above path is in fact a skip path: it is the shortest path that links row [1] to row [534]. Since the row numbers in a *skip path* are predetermined by their beginning and ending numbers, I'll denote this set of ordered numbers as $\{1 \rightarrow 534\}$. Similarly,

$$\{a \rightarrow b \rightarrow c\} = \{a \rightarrow b\} \cup \{b \rightarrow c\}$$

shall denote the union of 2 skip paths. Paths and skip paths (their row numbers) have interesting algebraic properties. In particular,

1. Every path is a composition of skip paths.
2. Every path includes the row numbers generated by the skip path with the same start and ending row numbers.

3.4.3 Source Row (Specific)

Once the hash of row [N] is published (committed to), the ledger owner may later reveal the contents of any row numbered less than or equal to N. If the revealed row is numbered r ($1 \leq r \leq N$), then the path shall include the full rows $\{1 \rightarrow r \rightarrow N\}$. The contents of the source row numbered r can then be revealed the "usual way" by matching the source row data with the full row's *input-hash*.

4. Other Properties

The properties enumerated above satisfy the definition of a commitment scheme. This next set of properties make the skip ledger a *useful* primitive. The focus here is on the properties of paths, the hash proof structures that enable partial reveals.

4.1 Logarithmic Succinctness

The skip ledger paths linking any set of row numbers are compact. The number of *full rows* in any skip path is of order **O** log n (where n is the *difference* of the last and first row numbers). Additionally, each row hash in a skip path on average encodes log n many hash pointers to previous rows: so, on average, the *Merkle tree* used to construct the hash of a full row contains **O** log n many previous-row hashes as its leaves. Thus each *Merkle proof* linking successive full rows in a skip path is of size **O** log (log n). Multiplying the average number of full rows with the average size of each row's Merkle proof linking it to the predecessor row in the path, the binary size of a skip ledger path is of order

$$\mathbf{O}\ (\log \log n) \log n \cong \mathbf{O} \log n$$

The justification for the "near equivalence" to **O** log n is that the first factor (log log n) grows extremely slowly and from a practical standpoint can be expected to never exceed 6 ($n = 2^{64}$).

In actuality, not all *link* proofs in the path structure are worth expressing as Merkle proofs: for a row numbered n, if n is not a factor of 16, then simply listing (or otherwise inferring) all the referenced row hashes (by row [n]) is more space efficient.

4.2 Ensemble Compressibility

Since paths from the same ledger share common lineage (and therefore common hashes), they take less space when archived together than each individually. This is because paths can be decomposed into (and recomposed from) their parts.

4.3 Composition

Some paths can be concatenated with other paths from the same ledger. In particular, given two paths from the same ledger for rows numbered a, b, and c

$$\{a \to b\} \text{ and } \{b \to c\}, \text{ with } a < b < c$$
$$a, b, c \in \mathbb{N},$$

a new path from a to c, $\{a \to b \to c\}$ can be composed from their "union" (concatenation). Generally, given any two *overlapping* paths from the same ledger

$$\{a \to c\} \text{ and } \{b \to d\}, \text{ with } a < b < c < d,$$
$$a, b, c, d \in \mathbb{N},$$

$$\exists\ b^{+} \in \{a \to c\} \cap \{b \to d\}, \text{ with } b \leq b^{+} \leq c$$

from which a new path

$$\{a \to b^{+} \to d\}$$

can be constructed.

4.4 Commit Provenance

If a ledger owner periodically publishes their ledger's commitment hash, they can also publish an *intermediate* commitment artifact, revealing the no. of rows in the ledger when that commitment was made, as well as a succinct proof linking the last commit to the previous commit. This is achieved using skip ledger paths, of course. Since paths are compact, a ledger owner may opt to publish a commitment as a path linking the latest committed row to the previously committed row.

4.5 Ensemble Commit Update

If paths from the same ledger are archived using properly overlapping sub-paths, then the addition of a single path (from the highest numbered row in the collection to a more recently committed row)

updates *all* the proofs (paths) in the archive to the path's highest numbered row. I.e. a single piece of compact data updates all existing proofs to the hash of a more recent commit.

This concludes the roundup of the properties of skip ledger proof structures. Paths have other interesting properties worth exploring (the algebraic properties of their full row no.s, for e.g.) but those are beyond the scope of this paper.

5. Representation

In principle, a ledger managed under this commitment scheme can just record the hash of the last row committed to an abstract skip ledger. In many niche cases this simple representation is appropriate. For example, if the lines in a log file are modeled as rows in a ledger and the log file is archived for its *potential* evidentiary value, then simply recording the skip ledger hash of the last line of the log file may be appropriate. If log files are rolled, then the (hash) state of the ledger can also be rolled from one log file to the next using a special compact structure called the *hash frontier*. However, none of these examples really concern this section.

What concerns us here is the data structures to use when it's time to partially reveal one or more row values against a committed hash. In order to tackle this problem, let us consider the case where we do not have advance knowledge about which row values will be revealed (at which row no.s) but we need to be able to generate such proofs (paths) quickly, at will. The `row_hash` function defined recursively above is immensely inefficient. Some form of persistent memo-ization is necessary to speed it up.

5.1 Skip Ledger Chain

One way to record skip ledger row hashes is to represent each row in fixed-size blocks. If each block only recorded the row-hash (the output of the `row_hash` function), then an efficient, memo-ized version of `row_hash` might be implemented using this sequence of blocks as a caching layer. However, the construction of paths (the hash proofs linking numbered rows) also requires the *input-hash* for the row (the output of the `input_hash` function which computes the hash of the source row numbered n). Depending on the data source this might also be an expensive, non-constant-time operation (finding the offset of a line no. in a log file, for example). To further speed up the construction of paths (and also to cover cases where the `input_hash` function is characteristically slow), we also record (and memo-ize) the input-hash of the row in each block. So if both the input- and row-hash functions both use SHA-256 (as used in the reference implementations [SL-REF][TC-REF] then each block representing a skip ledger row takes 64 bytes in the skip ledger chain.

Using the chain, individual row information is available in near constant time, and constructing paths takes $\mathbf{O}\ (\log d)^2$ block accesses, where d is the difference of the highest and lowest row number in the path). Note, the more skip pointers at a row number (the more trailing zeroes in the row

number’s binary representation), the more expensive it is to access linkage information for that row, but importantly, the more likely that row is included in a random path. An in-memory caching layer that prioritizes linkage information at row numbers with the most skip pointers, can ameliorate this log-squared performance penalty where it hurts most.

5.1.1 Chain Integrity

This section concerns detecting data corruption in the chain's blocks. It does not concern verifying that the committed source rows have not been tampered with. There are 2 classes of integrity-check.

Start-to-Finish

As in most "cryptographic chains" the integrity of the chain is verified by re-playing its blocks from start to finish. The source rows committed to the chain are not themselves verified in this "integrity-check": rather the row-hash recorded in each successive block is verified to be consistent with input-hash recorded in that block. On average, [the hash of] each skip ledger row (block) encodes (references) 2 previous row-hashes. An in-memory structure (the *hash frontier* previously mentioned in the context of rolling log files) obviates the need to read any block more than once during a run of the integrity-check.

Sampled Integrity

Every time a path is constructed from a skip ledger chain's blocks, the *derived* hash of the last numbered row in the path can be verified to match that recorded in the block at that row number. (The reference implementation makes this check.) This way, every path constructed from the chain, is also a sampled integrity-check: to the degree their skip path coordinates are random (as in their $\{a \rightarrow b \rightarrow c\}$ notation), the construction of paths over a chain can be seen as random-sampled integrity-checks. Since paths are the scheme's bread and butter, the more used, the less the need to verify the chain from start-to-finish.

5.1.2 Source Integrity

Another motivation for including the input-hash in the chain's blocks is that it allows us to distinguish chain corruption from *source* ledger corruption. The procedure for checking source integrity is to simply verify the recorded input-hash in each block matches that generated from the source ledger for each row no. from start-to-finish. This start-to-finish test, of course, can only be reliably performed by the ledger owner. However, every time a source row is revealed this check is also performed by the recipient of the revealed row. (As with paths, the reference implementation verifies the source row data matches the input-hash on packaging its proofs.)

6. Discussion

What to do with this primitive? Where to use it? This section compares and contrasts skip ledger with other techniques and approaches.

6.1 Comparison: Merkle Tree

The Merkle tree is a widely used primitive in a variety of settings. Skip ledger itself also uses Merkle trees internally. They share common properties, but depending on the setting, each enjoys advantages.

	Skip Ledger	Merkle Tree	Remarks
Succinct Proofs	✓	✓✓	Both skip ledger paths and Merkle proofs are **O** log *N* in size; however Merkle proofs are more compact.
Balanced Hash Tree	✓	✓✓	Merkle trees are perfectly balanced; skip ledger paths are slightly less balanced.
Commit Provenance	✓ (inherent in structure)		Skip ledger naturally links all historical states through its row hashes. Linking different Merkle roots, requires additional structures like hash chains, or skip ledgers themselves
Incremental Build / Commit	✓ (build-as-you-go)		Since the size of (no. of entries in) the collection must be known before they can be built, Merkle trees always have a final build step.
Use Synopsis			
Fixed-size Collections	✓	✓✓	Merkle trees are usually a better choice for fixed-size collections. However, if that fixed size is not known a priori, then skip ledger may still be a better choice (owing to its *incremental build* properties).
Append-only Collections	✓ (designed for this)		

Table 1. Skip Ledger / Merkle Tree Comparison.

6.2 Application to Blockchain

In most blockchains each block records the hash of the predecessor block in a specially reserved slot, as many bits wide as the hash function used requires. We'll examine how the hash recorded in that slot can be repurposed to instead use the skip ledger commitment scheme.

The gist of the strategy here is to identify and separate a block's application-specific data from its hash linking mechanism, and instead of calculating a block's hash the "usual way", use the hash of a block's "application-specific" data to compute a block's *input-hash* (in skip ledger terms) and "redundantly" record the (structured) skip ledger hash of the block itself in the the block's hash pointer slot. With proof-of-work chains like Bitcoin[Nak08], the hash-neutral nonce field will continue to not figure in the calculation of a block's hash (i.e. proof-of-work will not contribute to the block's *input-hash*); for chains operating under other consensus schemes, such as proof-of-stake, it may be appropriate to include proof-of-consensus data in the calculation of the input-hash.

Note skip ledger can be adopted retroactively. That is, once the chain's `input_hash` function is properly defined, the skip ledger commitment hash is computable for every existing block. The existing blocks can be augmented using a parallel chain that only records the skip ledger commitment hash at each block no. At 32 bytes overhead per block (using SHA-256), this would count as perhaps the smallest index built atop the chain. The auxiliary parallel chain ends right before the block number the commitment scheme is switched over to.

Starting from the block no. the scheme is adopted, blocks are referenced by their skip ledger hash (instead of how block-hashes are presently calculated), which, again, is redundantly recorded in the block itself. The proof-of-work puzzle (attempting to zero the hash of the block, computed the *old way*) remains unchanged.

6.2.1 Potential Advantages

Under skip ledger, any random existing block can quickly be verified to belong in the chain (as represented by the hash of any of its recent blocks) in sublinear time and space. This, in turn, may aid in tracking, verifying, or packaging off-chain proofs-of-provenance for a chain's tokens.

6.3 Application to General Ledgers

Business ledgers are usually maintained in relational databases. Smaller businesses, on the other hand, may simply maintain some of their "ledgers" in spreadsheets. Regardless where and how such ledgers are kept, businesses can ensure (and prove) their ledger data have not been corrupted by periodically committing to (recording) the hash of their ledgers.

Since many existing ledgers are in tabular form, the design of the `input_hash` function for such use cases is also an area of study. Areas of focus include:

1. How to define a ledger using a single SQL query (a SELECT statement often joining data from multiple tables for each logical row number).

2. How to compute a ledger row's input-hash from the composition of its individual cell hashes so that individual row cells may be redacted when revealing the contents of a row.
3. How to salt individual cell values so that their salted hashes resist rainbow attacks, frequency analysis, etc. with negligible space overhead. The prototyped solution uses a secret ledger-wide pseudo random byte sequence as seed to generate
    a. a row-number-specific pseudo-random salt (computed from the SHA-256 hash of the secret seed and the row number), and
    b. a cell-specific pseudo-random salt (computed from the SHA-256 hash of the row-specific salt and the cell's index / column number).

Under the "table salt" scheme (3) , the salt for every cell value is derivable from the row-salt and the cell's index (in the row), but not the other way around. When no values are redacted, only the row-salt needs to be revealed in the proof; if any cell value *is redacted*, then the row-salt *must not* be revealed and every revealed cell value is accompanied by the cell-specific salt.

6.4 Application to Data Streams

The commitment scheme finds use in the sharing and archival of large chunks of data (which may contain sensitive, or privileged information) for non-repudiation purposes. For such use cases, it is merely enough to calculate (and commit to) the [skip ledger commitment] hash of the last "row" or block in one pass.

For unstructured data streams, that is, when the commitment system's `input_hash` function has no prior knowledge of the stream's format, one may simply choose an `input_hash` function that merely partitions the stream as contiguous, fixed-sized blocks. This may be appropriate for video streams, for example.

From a practical perspective it may suffice to record (publish) the commitment hash of, say, every 1024-*th* block in the stream. This would allow any snippet to be verified against the commitment hash of the whole (as well as determine *where* it occurs in the stream). If the stream's commitments are periodically timestamped, the proof might also provide guarantees about *when* the snippet occurred.

6.5 Application: Timechain Protocol

Timechain [TC-REF] is an experimental REST based protocol for witnessing hashes. Here, a skip ledger chain models contiguous, equal-duration time-blocks since the chain's genesis. The chain's `input_hash` function computes the Merkle root of the hashes witnessed in that time-block. Users are expected to retrieve witness proofs promptly: the Merkle tree used to compute the block's input-hash is not kept indefinitely.

Historical Note

Earlier versions of the timechain did not use this commitment scheme. It developed the other way around. The design of skip ledger was first motivated by considering how best to construct the hash of an evolving ledger to be witnessed by the (old) timechain.

6.6 Every Receipt is From a Ledger

Every *modern* receipt is recorded in one or more ledgers. Gone are the days when an official document (receipt) might merely be imprinted with the King's seal and not recorded anywhere else. Thus if we annotate existing receipts with skip ledger commitments, proving at what row number[s] the ledgered receipt is recorded in, then the receipt can be efficiently linked (verified to belong) to all future commitments of the ledger.

6.7 Comparison: Blockchain

Blockchains also model ledgers. Since going mainstream, blockchain is now a toolkit many cutting edge applications must consider. Here the tradeoffs of the blockchain approach versus managing privately controlled ledgers maintained under the skip ledger scheme are examined at high level. But note, we're examining what is possible *in theory*: for example, if a blockchain (which *is* a ledger) can create tokens of value in its entries, then so too may privately controlled ledgers (backed by business processes operating under conventional legal regimes).

The table below contrasts the 2 approaches.

	Skip Ledger	Blockchain	Remarks
Tamper proof	✓	✓	
Governance	private	by consensus	
Legal Regime	contract / judicial	*code-as-law* (mostly)	
Decentralized		✓	
Ledger Rules	by convention / contract	by code	
Backward-compatibile	✓		
Easy-to-Model	✓		Conventional ledgers operate under more controlled, less adversarial environments than blockchain.
Off-chain verification	✓ (by design)		
Accountability	✓		Conventional ledgers are more accountable than decentralized blockchains.
Corrections / Revisions	✓		Conventional ledgers model corrections to existing entries with *new* "correction" entries (old entries are *never* edited, of course). By design, entries in blockchains are **final**.
Easy to evolve	✓		Conventional ledgers are easier to evolve than blockchains. In fact, skip ledger can *aid* in the *evolution* of a ledger: for example, back-end database schema changes can be verified not to break old commitments.
Tokenizable Parts	hypothesized	✓	

Table 2. Skip Ledger / Blockchain Comparison.

6.7.1 Source of Truth

The truth model under a pure commitment scheme approach for ledgers is fundamentally different from blockchain. The source of truth is the ledger's chain of commitments, not the actual contents of the ledger. With blockchain, the detailed contents of the ledger itself (the chain) are the source of truth. Moreover, the chain of commitments themselves are encapsulated in self-verifying compact proofs. The question whether or not to expose the chain of commitments (skip ledger chain) directly to users is an application-specific design choice. For example, the timechain protocol discussed earlier only exposes the commitment chain using skip ledger paths (named *block proofs*, in timechain's lingo.)

If blockchains can create tokens of value, can conventional ledgers also design and package tokens of value in their entries? Let us explore this issue using a perhaps contrived, over-simplified example.

6.7.2 Warehouse Ledger: a Hypothetical Application

A warehouse exposes its inventory as a public, mostly obfuscated ledger. At first, there are 4 types of record (row) in this ledger:

1. Item entering (stocked in) warehouse.
2. Item exiting warehouse.
3. Stocked item changing hands (ownership).
4. Correction to an existing record.

A live view of its commitments, in the form of an opaque skip ledger chain, is also made public. The warehouse only dispenses unredacted receipts to the privileged 3rd parties it transacts with. Type (1) or type (3) receipts from this ledger may function as proofs of ownership of items in stock. Entries in conventional ledgers often reference records in third party ledgers: for example, ownership-related entries in the warehouse ledger likely reference third party records. So too are the shipping-related entries in type (1) and type (2) records.

Note, regardless *how* and *where* an item's change-of-ownership is first recorded, the fact the warehouse provides verifiable receipts for when warehoused stock has changed hands, makes its *in-stock* receipts valuable. An extreme example of this might be by a legal judgement against the current owner of record; the "happy path", of course, is where both parties are in agreement (warehoused items sold or bartered).

Business is good at the warehouse. After a while, managers notice some customers are using their warehouse receipts as collateral for loans. They decide this is a good thing, but going forward, in addition to recording ownership, the warehouse ledger will also allow legitimate parties to also record contractual encumbrances on ownership. Accordingly, the warehouse introduces a new record type (call it type 3a) for recording that ownership in an already stocked item has become encumbered.

7. Closing Remarks

Skip ledger is a powerful addition to the toolbox. A number of implementation-specific, engineering details not covered in this paper are further explored in the reference implementations [SL-REF][TC-REF][MK-REF]. For the most part, these concern realizing efficiencies by developing a uniform set of tools and formats that both ledger owners and users (receipt recipients) know how to use in very general and broad settings. In particular, a binary archive format for bundling ledger proofs is prototyped. Because ledger entries often reference entries in *other* ledgers, each archive (called a *morsel*) packages proofs from *multiple* ledgers. For example, morsels package related timechain proofs (witness proofs of ledger hashes) much the same way as any other ledger.

References

[HS91] Haber, S., Stornetta, W.S. (1991). *How to time-stamp a digital document.* Journal of Cryptology 3, 99–111

[Mer78] Merkle, R.C. (1978). *Secure communications over insecure channels.* Communications of the ACM, 21(4)

[Mer82] Merkle, R.C. (1982). *Method of providing digital signatures*. US patent 4309569

[Lam81] Lamport, L. (1981). *Password Authentication with Insecure Communication.* Communications of the ACM, 24(11)

[Nak08] Nakamoto, S. (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System*, https://bitcoin.org/bitcoin.pdf

[Lam79] Lamport, L. (1979). *Constructing Digital Signatures from a One Way Function.* SRI International

[Mey23] Meyer, A. *Sok: Authenticated Prefix Relations – A Unified Perspective on Relative Timestamping And Append-Only Logs*. https://arxiv.org/abs/2308.13836

[Pug90] Pugh, W. (1990). *Skip lists: a probabilistic alternative to balanced trees.* Communications of the ACM, 33(6), 668-676.

[KBPB19] Kelsey, J., Brandão, L. T., Peralta, R., & Booth, H. (2019). *A reference for randomness beacons: Format and protocol version 2* (No. NIST Internal or Interagency Report (NISTIR) 8213 (Draft)). National Institute of Standards and Technology.

[SL-REF] Skip ledger reference implementation. https://github.com/crums-io/skipledger

[TC-REF] Timechain protocol reference implementation.
https://crums-io.github.io/timechain/overview.html
Sample deployment: https://crums.io

[MK-REF] Merkle tree implementation (skip ledger dependency).
https://crums-io.github.io/merkle-tree/