Network Working Group
Internet-Draft
Intended status: Experimental
Expires: 10 August 2026

T. Przygienda
T. Li
Juniper Networks

6 February 2026

ISIS Hierarchical SNPs
draft-prz-lsr-hierarchical-snps-01

Abstract

The document presents an optional new type of SNP called a Hierarchical SNP (HSNP). When feasible, it compresses traditional CSNP exchanges into a Merkle tree-like structure, which speeds up synchronization of large databases and adjacency numbers while reducing the load from regular CSNP exchanges during normal operation.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 10 August 2026.

Copyright Notice

Table of Contents

1. Introduction

The document introduces an optional new type of SNP called the Hierarchical SNP (HSNP). It compresses traditional CSNP exchanges into a Merkle tree-like structure [MERKLE], enabling faster synchronization of large databases and adjacency information while reducing the overhead of regular CSNP exchanges during steady state operation. By combining parallel flooding, CSNP exchanges, and HSNP-based compression, database resynchronization can be accelerated because fewer packets (not the entire CSNP set) need to be exchanged to fix inconsistencies. Using HSNPs also reduces unnecessary flooding and communication overhead, while still detecting mismatched fragments more efficiently than full CSNP exchanges.

To maintain a consistent framework, we initially treat each CSNP entry for an LSP as equivalent to a "as good as perfect", though rather long, Merkle hash. In other words, in the first step, the LSP ID, sequence number, and checksum act as a "perfect fragment hash". We also include the fragment's PDU length to add more entropy in the

next step. The CSNPs then serve as the "bottom" of the Merkle tree conceptually speaking. Next, we compute a Fletcher checksum over this data to create a shorter fragment-level merkle hash. Those hashes are never transmitted in packets, since doing so would effectively duplicate CSNP functionality. However, the hash that combines all fragment hashes of a node becomes a "node Merkle hash". Groups of such node hashes can then be summarized again into a "node range Merkle hash", created by hashing the individual node hashes together. For this final step, we switch from Fletcher checksums to a different hashing method, as explained in Section 4. The resulting hierarchy of hashes enables validation of large LSDB synchronizations using far fewer packets than relying solely on CSNPs. Although these hashes summarize ranges recursively, the resulting exchange, depending on range mismatches, resembles a [SKIP]-like process rather than maintaining a fixed tree structure in every exchange.

This document limits itself for practical purposes to LSDB sizes of the order of 1E6 fragments and further considerations necessary to prevent overly garrulous exchanges of hashes covering smaller and smaller sets of fragments. More details on the targeted IS-IS envelope can be found in Section 8 and further considerations revolving around these assumptions are summarized in Section 9.

2. Dynamic Partitioning

In practical terms, the most interesting problem is figuring out how to divide the database into groups of leaf nodes (each representing a set of fragments) that both compress the data effectively compared to standard CSNPs and remain as stable as possible. If the fragment boundaries within each hash keep changing, neighboring systems will need to recalculate their hashes instead of reusing cached ones from their own Merkle tree, which adds unnecessary computation. The subdivision should also create enough "bins" to handle any distribution of fragment IDs across the network. This helps avoid pathological cases where all fragments might end up in a single hash, such as when e.g. a hashing function degrades. Ideally, when a hash mismatch occurs, it should take only one or two packets, with hashes for smaller fragment groups or regular CSNPs, to fix the difference. Reducing I/O and/or computation directly improves how quickly the systems synchronize.

To begin, in IS-IS networks we can fit just under 100 PSNP entries into the typical 1500-byte MTU frame. This is the consequence of each CSNP entry including the Node ID, Fragment number, Sequence number, Checksum, and Lifetime fields -- totaling 7 + 1 + 4 + 2 + 2 = 16 bytes per entry.

Fletcher hashes, as shown in Figure 2, take up 6 + 6 + 6 = 18 bytes each, which means that about 70 such hashes fit into a single packet. A more detailed explanation of why a 48-bit Fletcher checksum and the resulting hash sizes were chosen is provided in Section 9.2.

These limits form the basis for the recommended partitioning and packing strategies discussed later.

At the lowest compression level, it is optimal to generate a single CSNP packet on a mismatch in a hash. To achieve this, the first-level hashes should initially group about 80 LSP fragments together, with exceptions handled later. There is no need to maximize this initial packing.

As the LSDB grows, it is better to leave some flexibility ("slack") in how fragments are grouped. This increases the likelihood that both sides of an adjacency will maintain the same leaf-level packing, even during flooding transitions, and prevents the ranges from shifting constantly.

The packing process always places all fragments belonging to the same system and its pseudonodes within a single node Merkle hash. This hash may occasionally exceed the recommended size of 80 fragments. When it comes to node range hashes a hash is considered "full" when adding the next system’s fragments would go beyond this limit. Unless, of course, the leaf is still empty.

At that point, any range hash can be folded into a higher-level, less specific range hash. However, whenever a mismatch occurs at any level, the process must disaggregate the hash and send the corresponding smaller, more specific range or node hash instead.

2.1. Denser Packing and Repacking

To talk meaningfully about node range hashes, we will refer to hashes that cover a wider range of nodes as less specific, and to those covering only a subset of that range as more specific.

Instead or in addition to the "first level" packing a system can decide to pack "more densely". In such a mode, the HSNP may include hashes that cover a much larger range than the first-level hashes. How this denser packing is implemented is left up to the specific implementation.

A good general approach is to increase packing density in parts of the database that have not changed, where no hash mismatches have been observed, or when it is reasonable to assume that the neighbor already holds a mostly synchronized database.

As a secondary consideration, it would be useful for efficient, cache-based implementations for both sides to agree on the ranges of Merkle hashes advertised. This would make caching of merkle nodes much more effective. However, while this idea seems viable in theory, implementing it across a large number of interfaces would effectively require a global synchronization protocol - something impractical in a network where nodes are constantly adding, removing, and updating fragments asynchronously. These ongoing changes continually affect what the "optimal" hash ranges are. And with enough churn, such a range negotiation protocol might never converge at all.

Alternatively, providing a fast way to reconstruct the internal Merkle hash for a mismatched range could reduce the need for perfect range alignment. For example, in the proposed packing scheme, nodes always agree on system ID boundaries. By maintaining a Merkle hash per system ID, a node can quickly recompute the required hashes whenever received ranges differ from its cached ones - even in networks with a large number of nodes.

It is still highly preferable for advertised Merkle hash ranges to align on system ID boundaries as much as possible - especially at the top level. Under stable conditions, these top-level Merkle hashes significantly reduce the amount of CSNP exchange required, minimizing both packet volume and processing overhead.

Even though a fully stable network could, in theory, be represented by a single hash covering the entire LSDB, doing so is neither desirable nor beneficial. Since an HSNP packet must be sent anyway, it is much better to fill it with around 70 node range hashes. This approach limits the amount of decompression required if a collision occurs within one of those ranges and also reduces the risk of hash collisions, as discussed in Section 9.2.

In summary, a node should avoid compressing beyond the point where a single HSNP covers the entire database. Ideally, one HSNP should contain at most about (MTU / Node Range Hash Entry Size) hashes - or fewer - to keep collision probabilities low, as described in detail in Section 9.2.

3. Hash Function for a Fragment

Each fragment generates a 48-bit Fletcher checksum, which is a straightforward extension of the existing 32-bit standard. For performance reasons, this Fletcher variant replaces the modulo operation with the common approximation of a simple bit shift. The choice of a 48-bit checksum is based on the analysis provided in Section 9.2.

Appendix A provides reference implementation of algorithm building the 48-bits fletcher checksum that serves as hash of fragments.

In case the fletcher checksum generates a zero value the value MUST be replaced with constant of value 1.

To validate correctness of implemented fletcher reference hash given in Figure 1 can be used.

```
Node ID: 0101:0101:0000.01
Fragment#: 01
Sequence#: 0001
ISIS Checksum: $0001
ISIS PDU Length: 512
Hash: $090E0E020404
```

Figure 1

4. Fast, Incremental, Self-Inverse Hashing Function for Fragment Ranges

Since large-scale deployments must compute significant numbers of hashes over sets of frequently changing fragments, it is highly desirable to use a specialized hash function that supports fast incremental updates when fragments are added, removed, or when their checksums change.

Once hashes are built over sets of fragments, it is desirable to support very fast splitting and merging of such sets, especially when two hashes differ in which fragments they contain.

Deeper considerations on such hashes can be found in [HASHES] but our design space is simplified due to irrelevance of security involved.

The hash for a set of fragments is computed using a very fast XOR over their Fletcher hashes. This makes it straightforward to update the hash when a leaf is added, removed, or its checksum changes. As a result, less specific ranges can quickly derive their hash by XOR'ing the hashes of all included, more specific ranges, when those are available.

5. Procedures

5.1. HSNP Support Negotiation

IIH of nodes supporting this extension MUST include in IIH a new TLV that will indicate support for reception of HSNPs. All nodes on the adjacency MUST advertise the TLV on their IIHs, otherwise HSNPs are not used. Observe that a node may very well just receive and process the HSNPs and answer them always with CSNPs necessary although this is obviously less beneficial than fully supporting sending and receiving of HSNPs.

5.2. Advertising and Receiving HSNPs

Advertising standard CSNPs is extended with HSNP advertisements when this feature is supported. Since both CSNPs and HSNPs carry range information in their headers, they can be freely mixed, depending on which level of fragment "compression" best fits the situation. In practice, sufficiently specific range mismatches will naturally fall back to CSNP exchanges or flooding to resolve remaining differences.

The ranges MUST be sorted based on Range Start System ID. The ranges MAY overlap albeit it is highly undesirable.

Any node IDs that are not covered by the ranges in a packet - either because there are gaps between the advertised ranges, or between those ranges and the HSNP’s Start and End System IDs - MUST be treated as missing. Consequently, if a node detects that it holds Merkle hashes for LSPs that are not covered by a received HSNP, it MUST behave as it would in the same situation with a CSNP, namely by flooding the missing LSPs. Alternately, instead of leaving a "hole" in the packet range, a HSNP can be included with Merkle hash being set to 0 (which will generate a guaranteed miss) and interpreted as nodes in the range being present but not covered by HSNP compression anymore. The node sending such an entry SHOULD send according CSNPs before the according HSNP packet.

As with CSNPs, an HSNP whose first range covers the first node in the database MUST use 0000.0000.0000 as the start system ID in its packet range so that missing nodes can be detected. The same rule applies at the other end of the database: an HSNP whose range covers the last node MUST indicate this in a way that allows detection of any missing trailing nodes.

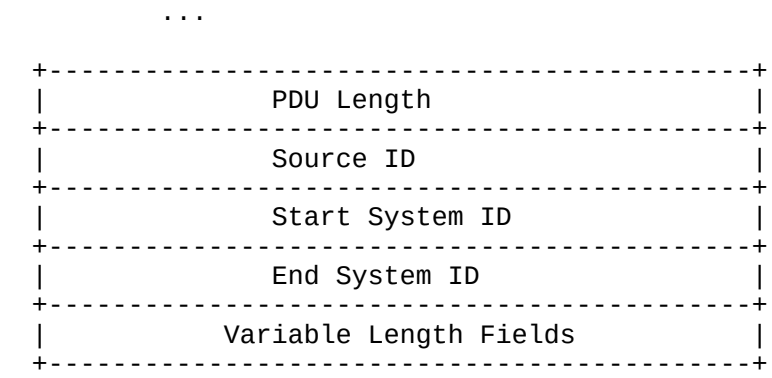
When a node receives an HSNP where any of the contained hashes does not match after recomputation or comparison, it MUST immediately send HSNPs with Merkle hashes covering the mismatched ranges. These new hashes MUST be more specific than the range where the mismatch occurred.

Alternatively, instead of more specific HSNP hashes, a node MAY choose to send corresponding CSNPs, PSNPs, or flood the mismatched LSPs directly. Sending CSNPs or flooding immediately may be preferable when the mismatch affects only a small number of LSPs.

If there is a mismatch - or no computation available - for a hash covering just a single node (with its pseudonodes), or for a hash spanning fewer fragments than a full CSNP PDU, then a CSNP MUST be sent. Alternatively, the node MAY choose to flood those specific fragments directly instead.

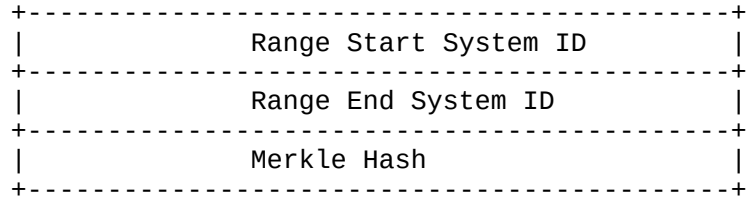
6. HSNP PDU Format

The HSNP PDU format closely follows the CSNP format. Instead of CSNP entries, it carries the corresponding Merkle hashes - which cover exactly the same fragments that would appear in CSNP packets. The Start and End System IDs exclude pseudonode bytes, as those are implicitly included within the ranges.



The Start and End System IDs use the standard ID length and indicate the range of fragments covered by the HSNP, just like CSNPs do. The key difference is that all pseudonodes of the systems within this range are implicitly included. Both the Start and End System IDs are inclusive, meaning fragments from both endpoints are part of the range.

The variable length fields are a sorted sequence of Node Range Hash Entries in the following format.



Range of LSPs that are included in the hash. Range Start and Range End System ID are inclusive, i.e. the fragments of both the start and the end system ID are contained within the range.

Merkle hash consists of 6 bytes of the 48-bit computed hash of all fragments covered by the range.

This makes an entry in typical deployment scenarios 6 + 6 + 6 = 18 bytes long and hence about 70 hashes fit into a typical MTU .

Ranges MUST be sorted on Start System ID.

7. Example

An example will clarify things further. Consider an LSDB with 512 nodes, each having a system ID of 1000.0000.00<2 digits node-id> and holding 32 fragments numbered 0 - 31. We skip uneven node identifiers to create intentional "holes" in the numbering. The pseudonode byte is treated simply as part of the system ID, since it doesn’t affect the scheme itself.

In a stable state, reasonable compression can deliver 128 "first-order" leaves - each containing fragments from 2 systems (64 fragments total) - requiring roughly 512 / (2 * 70) ~ 4 packets. The first of these "first-order" packets would look approximately like this:

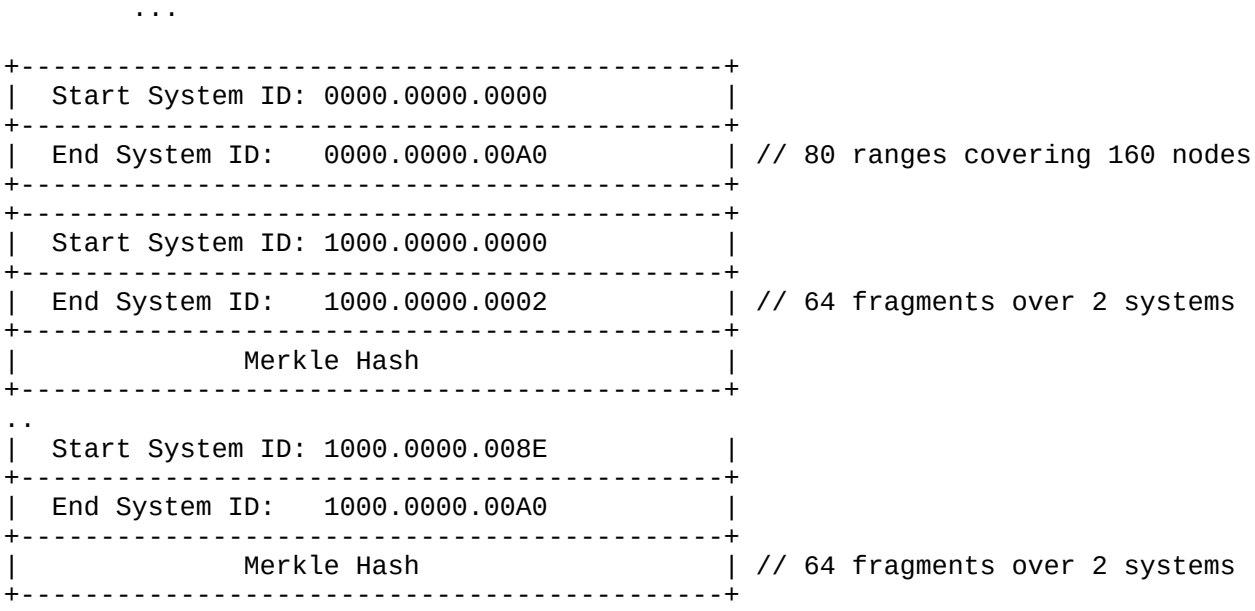


Figure 2

Based on local decisions, a node can further compress HSNPs until - in the most extreme case - it sends just one packet full of hashes. In our example with 512 nodes, this divides them across 70 hashes (assuming equal fragment counts for simplicity), resulting in about 8 nodes per hash (equivalent to 8 * 32 fragments). The resulting packet would look like this:

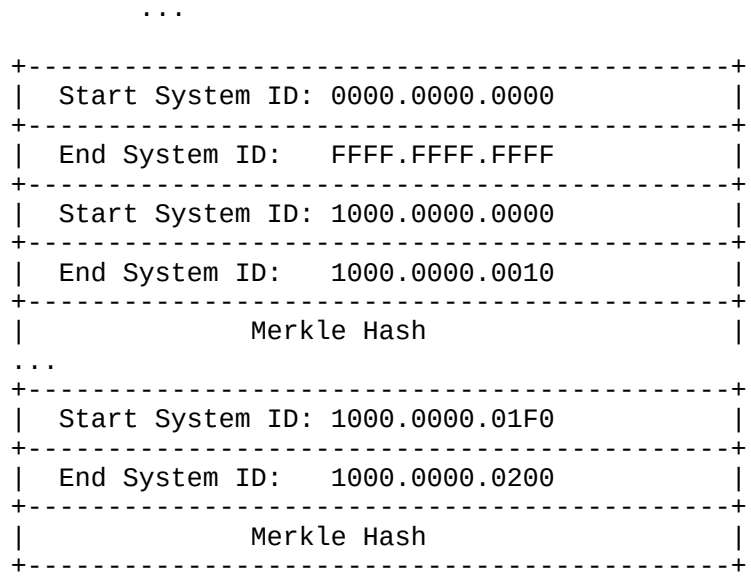


Figure 3

8. IS-IS Scale Envelope Considerations

HSNPs provide negligible benefit in small networks with only tens of nodes and hundreds of fragments. Perfect flooding would theoretically suffice at any scale (though history shows this is too optimistic), and even CSNPs represent under such assumptions protocol overhead only - yet they have significantly contributed to IS-IS stability in real deployments.

As networks grow larger, the costs of link flaps, node restarts, and periodic CSNP exchanges increase substantially. HSNPs can significantly extend IS-IS scalability in these scenarios.

To push IS-IS to its practical limits, we must account for flooding rates driven by fragment refreshes and LSDB synchronization. Path computation impacts can be deferred - bare pathological scenarios - by using dampening and parallelization, and thus we focus on realistic scenarios in the order of 50,000 nodes and 1 million fragments.

At maximum configured lifetime, this generates ~15 packets/second per interface from refreshes (1M/65K fragments/second), plus ~10,000 CSNP packets for full LSDB sync.

Achieving a modest 120-second sync requires ~80 CSNPs/second, and across 16,000 interfaces, that represents a peak of 1.5 million packets/second - far beyond current fast-flooding capabilities. We disregard here further techniques like [ID.draft-ietf-lsr-distoptflood-11], especially since they do not improve CSNP scale.

With maximum HSNP compression, however, sync overhead drops to roughly one additional packet beyond refresh flooding, leaving ~250,000 packets/second (15 * 16K) to be handled by fast flooding and flood reduction techniques - a challenging but feasible target.

The considerations above make it clear that combining fast flooding, flood reduction, and HSNP features will be essential in extending IS-IS scalability as deployments continue to grow larger.

9. Further Considerations

9.1. Maximum Advisable Hash Coverage

Although the mechanism can theoretically use a single Merkle hash to represent an arbitrarily large database, such an approach is not advisable. Instead, it is far preferable to limit hash coverage so that at minimum one full HSNP packet is required.

In practice, limiting compression so that a maximum of about a dozen HSNP packets covers the entire database is usually sufficient. For example, a single maximally compressed HSNP packet for a 1,000 - fragment database covers ~140 fragments per hash. Allowing for more HSNP packets (e.g., 10 instead of 100 CSNPs) still provides a 10x compression factor, reduces disaggregation needs during LSDB changes, and further lowers the already negligible collision risk (which in 1K sized LSDB is however vanishingly small with a single hash already).

9.2. Hash Collision Probabilities

Even with CSNPs or PSNPs, IS-IS has a corner case where LSDB synchronization can fail - particularly during node restarts. In simple terms, if a new fragment has the same sequence number and different content but an identical 16-bit Fletcher checksum, the collision goes undetected until the fragment expires.

Assuming Fletcher checksums are uniformly distributed (even with minor content changes), the collision probability for that case is 1/2**16 ~ 0.0015%. This baseline enables meaningful comparisons with HSNP Fletcher collision probabilities.

HSNP uses 48-bit Fletcher checksums over what is essentially PSNP data for a fragment plus the fragment's IS-IS length. The key concern is the probability that two fragments - covered by the same hash - generate the same Fletcher checksum simultaneously. This would cause both fragments to "disappear" due to the XOR checksum nature. Collisions occurring for fragments in different node range hashes are irrelevant.

One might argue that XORing different sets of hashes could produce the same result, but the probability of two distinct sets having identical modulo-1 sums across all 48 bits is vanishingly small (~1E-15). This scenario is not considered further.

Collision probability analysis is complex for the general case, though the birthday paradox gives a rough estimate of 0.18% collision likelihood for 48-bit hashes in a 1M - sized set. While 64-bit hashes would intuitively reduce this to ~0.0000027%, we instead rely on extensive simulations that mirror real-world conditions.

These simulations model 50,000-node networks with 1M fragments, assuming node IDs differ by only 3 bytes, maximum fragment lifetimes, random protocol checksums on fragment refresh, and packet length changes in just 5% of refreshes to reflect network stability. Results are derived from 32 networks running for 2 years each.

Across 64 years of simulation and resulting 36 billion refreshes of all fragments, we observed only 200 - 300 collisions on the hash of the whole set - a probability of 6E-9, or roughly 3 occurrences per year. These collisions have an average lifetime of about 10 hours. This rate is orders of magnitude lower than the birthday paradox prediction of 0.1%, likely because node IDs act as a consistent "salt," effectively pre-partitioning the probability space.

The situation becomes more practically relevant when considering collisions within a single highest-compression packet containing 70 hashes. This roughly halves the collision rate, leading to about 1 collision per year in such a large network. If compression is instead limited to 70 HSNP packets (rather than maximum compression), the rate drops further to approximately 1 collision per 10 years.

Surprisingly, switching to 64-bit hashes reduces total collisions by only about 50%, and under maximum compression (single full HSNP packet), the results are actually measurably worse. This appears to be because 64-bit collisions tend to cluster more closely together in the database - a phenomenon we currently lack an explanation for.

A highly conservative (not to say paranoid) implementation can simply monitor the LSDB for colliding Fletcher hashes among fragments and exclude them from the same HSNP hash. This forces receiving nodes to use separate collision-free hashes instead. Such an approach completely eliminates any risk of synchronization misses when using HSNPs.

Other techniques are possible, such as slowly walking the database and sending CSNPs. However, for a 1M-fragment database that generates 10,000 such CSNP packets, the chance of this detecting a collision during its 10-hour window is likely extremely small.

9.3. Impact of Packet Losses

Hashes covering large numbers of fragments are more vulnerable to packet losses, as each lost packet affects a much larger portion of the LSDB during synchronization. Implementations can choose HSNP node ranges freely, but should balance maximum compression against "good enough" compression that reduces both collision risk and vulnerability to unavoidable packet drops.

9.4. Decompression and Caching/Comparison Optimizations

As mentioned earlier, nodes can use various strategies to accelerate decompression. For example, LSPs missing from HSNPs (those not covered by any ranges) are clearly absent and can be immediately reflooded. Similarly, small mismatched Merkle ranges can trigger immediate CSNPs, PSNPs, or direct flooding.

Caching of hashes can be applied at many levels. The simplest and most useful approach is maintaining a hash for all fragments of a node and its pseudonodes, though other granularities work equally well. Even when adjusting for changes - such as receiving a range < A - B > while having cached < A - B & next-after-B > - the cached hash can be quickly updated by simply XORing out the next-after-B node Merkle hash.

10. Security Considerations

TBD

11. IANA Section

TBD

12. Contributors

TBD

13. Acknowledgement

People have been talking about "compressing CSNPs" for a very long time, reportedly going back to when Radia Perlman and an insomniac Dave Katz were walking the halls discussing it. Recent attempts to scale the protocol much further have made it worthwhile to turn this idea into a standardized, practical engineering solution.

Les Ginsberg identified several unresolved issues and contributed alternative ideas to the draft.

14. References

14.1. Normative References

[HASHES] Phan, C.-W., "Security considerations for incremental hash functions based on pair block chaining", 2006.

[MERKLE] Merkle, R.C., "A Digital Signature Based on a Conventional Encryption Function", 1988.

[SKIP] Phan, C.-W., "Skip lists: A probabilistic alternative to balanced trees", 1990.

14.2. Informative References

[ID.draft-ietf-lsr-distoptflood-11] White et al., R., "IS-IS Distributed Flooding Reduction", October 2025, <https://www.ietf.org/id/draft-ietf-lsr-distoptflood-11.txt>.

Appendix A. Reference Implementation of Fragment Hashing

```
<CODE BEGINS>
pub fn fletcher_hash(fragmentid: &SharedFragmentID, fragmentcontent: &FragmentContent)
 -> HSNPHash {

    let nid = fragmentid.node.node_id().0;
    let pnodebe = fragmentid.pnode.0.to_be_bytes();
    let seqnrbe = fragmentcontent.seqnr.0.to_be_bytes();
    let fragmentnrbe = fragmentid.fragmentnr.0.to_be_bytes();
    let csumbe = fragmentcontent.isis_checksum.0.to_be_bytes();
    let lenbe = fragmentcontent.isis_pdu_length.0.to_be_bytes();

    let rotate_in_primary = nid.iter().chain(
```

```
            csumbe.iter().chain(
                seqnrbe.iter().chain(
                    fragmentnrbe
                        .iter()
                        .chain(lenbe.iter().chain(pnodebe.iter())),
                ),
            ),
        );

        // takes an iterator over u8 slice and return BE u16 stream padded with a final ze
ro if
        // not. necessary for fletcher to work
        pub struct U8ToU24BePadded<'a, I>
        where
            I: Iterator<Item = &'a u8>,
        {
            iter: I,
        }

        impl<'a, I> U8ToU24BePadded<'a, I>
        where
            I: Iterator<Item = &'a u8>,
        {
            pub fn new(iter: I) -> Self {
                U8ToU24BePadded { iter }
            }
        }

        impl<'a, I> Iterator for U8ToU24BePadded<'a, I>
        where
            I: Iterator<Item = &'a u8>,
        {
            type Item = ux::u24;

            fn next(&mut self) -> Option<Self::Item> {
                let b0 = self.iter.next()?;
                let b1 = self.iter.next().unwrap_or(&0);
                let b2 = self.iter.next().unwrap_or(&0);
                let v = u32::from(*b0) | (u32::from(*b1) << 8) | (u32::from(*b2) << 16);
                Some(ux::u24::new(v))
            }
        }


        match LIBRARY_HSNP_SIZE {
            HSNPSize::_48Bits => {
                let mut u24buffer: [ux::u24; 6] = [ux::u24::MIN; 6];
                U8ToU24BePadded::new(rotate_in_primary).collect_slice_checked(&mut u24buff
er[..]);
                fletcher::calc_fletcher48(&u24buffer).into()
```

```
          }
   <CODE ENDS>
```

Authors' Addresses

Tony Przygienda
Juniper Networks
Email: prz@juniper.net

Tony Li
Juniper Networks
Email: tli@juniper.net