Formal Verification of Authenticated, Append-Only Skip Lists in Agda

Victor Cacciari Miraldo*
Dfinity Foundation
The Netherlands
victor.miraldo@dfinity.org

Harold Carr
Oracle Labs
United States
harold.carr@oracle.com

Mark Moir
Oracle Labs
New Zealand
mark.moir@oracle.com

Lisandra Silva*
Inesc Tec
Portugal
lisandra.m.silva@inesctec.pt

Guy L. Steele Jr.
Oracle Labs
United States
guy.steele@oracle.com

Abstract

Authenticated Append-Only Skiplists (AAOSLs) enable maintenance and querying of an authenticated log (such as a blockchain) without requiring any single party to store or verify the entire log, or to trust another party regarding its contents. AAOSLs can help to enable efficient dynamic participation (e.g., in consensus) and reduce storage overhead.

In this paper, we formalize an AAOSL originally described by Maniatis and Baker, and prove its key correctness properties. Our model and proofs are machine checked in Agda. Our proofs apply to a generalization of the original construction and provide confidence that instances of this generalization can be used in practice. Our formalization effort has also yielded some simplifications and optimizations.

CCS Concepts: • **Software and its engineering** → **Formal methods**; • **Theory of computation** → *Data structures design and analysis*; • **Security and privacy** → *Distributed systems security*.

Keywords: Agda, Authenticated data structures, Blockchains, Formal verification, Skip lists

ACM Reference Format:
Victor Cacciari Miraldo, Harold Carr, Mark Moir, Lisandra Silva, and Guy L. Steele Jr.. 2021. Formal Verification of Authenticated, Append-Only Skip Lists in Agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3437992.3439924

*Work performed while an intern at Oracle Labs.

CPP '21, January 18–19, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8299-1/21/01…$15.00
https://doi.org/10.1145/3437992.3439924

1 Introduction

Decentralized technologies—such as blockchains [11]—that enable reliable coordination among parties that do not trust each other are of increasing interest and importance. It is common to maintain a self-authenticating *log* using *hash chaining* [18]: each entry (e.g., block or transaction) includes a cryptographic hash based on its own contents as well as on the previous entry. This technique ensures that a dishonest participant can "rewrite history" without detection only if it can find a collision for the cryptographic hash function used; it is a standard assumption that this is infeasible for a computationally bounded adversary.

Participating in consensus to add more entries to the log typically requires knowledge of the current "state", which is a function of all previous transactions. However, downloading and verifying the entire history and directly computing the state is too slow and expensive for many purposes, such as bringing a new participant online quickly.

One option is to provide a state and a quorum of signed digests for that state to a new participant p. We could further enable p to download only parts of the state that it needs by using more structured state types [10]. Although this enables p to participate in consensus for appending *new* blocks to the chain, receiving and verifying the state at index j does *not* enable p to verify claims about transactions at indexes $i < j$ without significant work. This is because hash-chaining typically used in blockchains is *linear*: to confirm a claim that a particular transaction is at index i, p must fetch and verify *all* transactions between i and j.

By instead using an *Authenticated Append-Only Skip List* [6] (AAOSL) to represent the log, an existing participant (the *prover*) can provide to a new participant (the *verifier*) an *advancement proof* from some previous log entry (perhaps the initial—or *genesis*—entry) to the new root digest that the verifier has obtained from a quorum of participants. The verifier can then receive a *membership proof* containing a

claim about a previous log entry, and confirm that the claim is consistent with the log on which the previously verified advancement proof was based. The new participant can also construct advancement proofs and/or membership proofs pertaining to new log entries; thus it can assist another new participant to bootstrap in future. Together, these ideas enable dynamic participation without requiring any participant to possess all historical information.

To provide this functionality efficiently, the hash stored in each log entry is made to directly depend not only on the previous entry but also on selected entries further back in the log. The "hops" back to these previous entries are arranged so that advancement and membership proofs are logarithmic in the length of the log. As a result, these ideas enable dynamic participation that can be sustained over a long period of time, because the work required for a new participant to join grows only logarithmically with the length of the log.

These advantages come at the cost of using mechanisms that are significantly more complicated than linear hash-chaining. Therefore, a formally verified model is paramount before using these techniques in practice.

Our primary contribution is a formal model of a class of AAOSLs and proofs of important properties about them in the Agda language [13]. Moreover, we prove that the original AAOSL [6] is an instance of this class and, consequently, enjoys the same properties. Finally, our formal verification work enabled us to simplify some aspects of the original AAOSL, such as the encoding of advancement proofs and the definition of authenticators.

We provide some background in Section 2 and then present a specification of the original AAOSL [6] in Section 3. In Section 4, we present our Agda model of a class of AAOSLs, proofs of key properties about them, and our proof that the original AAOSL is an instance of the class. We discuss related and future work in Section 5, and conclude in Section 6.

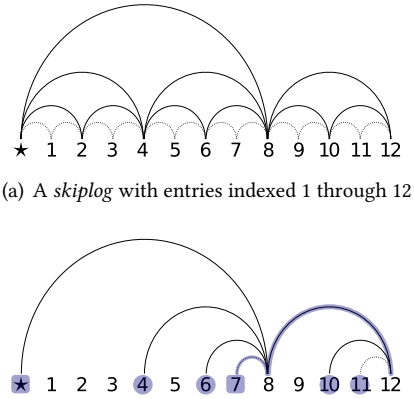
2 Background

To illustrate traditional linear hash chaining using Haskell, we define the following datatype and type synonyms.

```
data Auth a = Auth a Digest      type Log a = [Auth a]
```

Auth a—the type of authenticated values of some type *a*—comprises a value of type *a* and its *Digest*. We assume (via a type constraint *Hashable a*) that there is a function *hash* that receives an *a* and returns a fixed-length bytestring, which we call a *digest* (or *hash*).

The "genesis" log is represented as $[\,Auth\ a_\star\ h_\star\,]$, where $a_\star$ is agreed in advance and is used only to compute $h_\star = hash\ a_\star$; we sometimes refer to index 0 as $\star$. The *append* function illustrates the hash chaining technique.



(a) A *skiplog* with entries indexed 1 through 12

(b) The direct dependencies of h_8 and h_{12}, needed to verify a proof of advancement from h_7 to h_{12} assuming h_7 is already known and trusted.

Figure 1. SkipLog illustrations

```
append :: (Hashable a) ⇒ a → Log a → Log a
append x [ ] = error "Log not initialized"
append x (Auth y dy : l)
  = Auth x (hash (hash x ++ dy)) : Auth y dy : l
```

An initialized log comprises a head *Auth y dy* and a tail *l*. When appending *x*, the new log is constructed by hashing *hash x* concatenated with the latest hash in the log, *dy*. We chose to present *append* as adding an element to the *front* of the list for conciseness. An implementation that adds elements to the *tail* of a list would use the same hash chaining: the hash for the element at position $i+1$ depends exclusively on the hash of the element at position i. Whether we interpret position 0 to be the leftmost or the rightmost element of the list is irrelevant for the remainder of our formalization.

With a traditional linear log, a client that knows the j^{th} log entry can verify the i^{th} entry for $i < j$ only by recomputing each digest for entries $i+1$ to j to confirm that the recomputed digest for j matches. This is inefficient and requires possession of all entries between i and j.

An AAOSL [6] stores a (partial) log linearly as usual, but enables verifying earlier contents in the log, or verifying that a recent segment of the log builds on a known previous log, by verifying only a logarithmic amount of data.

3 The SkipLog

In traditional skip lists [16], for each inserted index, a "height" is probabilistically chosen, and for each level up to that height, a "skip pointer" to the previous index with a height of at least that level is stored.

A *skiplog* is an append-only variant, enabling the use of *deterministic* skip pointers per power of two, as seen in Figure 1(a). These skip pointers define a *dependency relation* (or *hop relation*) between indexes, and thus a *dependency graph*.

When traversing the list from index 12 to index 4, the shortest path is to take the skip pointer at level 3, from 12 to 8, then the one at level 3, from 8 to 4. A skip pointer at level l skips 2^{l-1} entries back. The *maximum skip level* for an index $i > 0$ is defined as $k + 1$ where k is the largest integer such that 2^k divides i. We use the type synonyms *Index* and *Level* (both *Int*s) to make the role of these values evident.

```
maxLvl :: Index → Level
maxLvl 0 = 0
maxLvl j = 1 + go j
  where go j = if even j then 1 + go (j `div` 2) else 0
```

The maximum skip level of a given index is exactly the number of other indexes it depends on. Therefore, the maximum skip level of index 0 is defined as 0, indicating that there is no skip pointer from 0. In our skiplogs, the hash of each entry is made to depend on all the entries one skip pointer away from it.

Appending an entry is similar to the description in Section 2. This time, however, we change how the hash of the appended entry is computed, ensuring that it depends directly on every log entry one skip pointer away from it. Thus, the hash for log index 8 depends on entries for 7, 6, 4, and ⋆.

```
append :: (Hashable a) ⇒ a → Log a → Log a
append x log = mkauth x log : log
```

The *auth x log* call computes the *authenticator* [6] of x at the new position in the log. The *authenticator* for position j is a hash computed using the hash of the data to be stored at j and the hashes of the entries one skip pointer back from j, namely entries at indexes $j - 2^{l-1}$ for $l = 1, 2, \ldots,$ *maxLvl j*.

```
mkauth :: (Hashable a) ⇒ a → Log a → Auth a
mkauth a log =
  let j    = length log
      deps = [lookupHash log (j − 2^(l−1)) | l ← [1 .. maxLvl j]]
  in mkauth a (auth j (hash a) deps)
```

The *lookupHash log i* function returns the authenticator for the entry at position i and the *auth* auxiliary function assembles the necessary data into a single hash. This hash is determined by hashing the concatenation of the *partial authenticators* [6] of each dependency in *deps*. In the original presentation, *partial authenticators* are computed by hashing the current index, level, data, and the authenticator of the dependency, illustrated by the following pseudo-Haskell:

```
auth :: Index → Digest → [Digest] → Digest
auth j datumDig lvlDigs = hash (concat
  [hash (encode j ++ encode 1 ++ datumDig ++ lookup 0 lvlDigs)
  ,hash (encode j ++ encode 2 ++ datumDig ++ lookup 1 lvlDigs)
  , ...
  ,hash (encode j ++ encode l ++ datumDig ++ lookup (l − 1) lvlDigs)
  ]) where l = maxLvl j
```

Translating the pseudo-Haskell above to working Haskell yields:

```
auth :: Index → Digest → [Digest] → Digest
auth j datumDig lvlDigs =
  let pauth lvl lvldig = hash (encode j ++ encode lvl
                                        ++ datumDig ++ lvldig)
  in hash (concat (zipWith pauth [1 . .] lvlDigs))
```

The *encode* function encodes data in a *ByteString*, facilitating concatenation and hashing with the rest of the data. As in Section 2, the authenticator for the last position in the log depends (directly or indirectly) on the authenticators of every prior position and thus provides a "cumulative" hash of the whole structure.

We started our work using the original *auth* definition [6] shown above. However, our formalization effort revealed that any definition of *auth* that is injective in its second and third parameters suffices. That is, any *auth* such that $auth\ j\ h_1\ ds1 \equiv auth\ j\ h_2\ ds2$ implies $h_1 \equiv h_2$ and $ds1 \equiv ds2$ (modulo hash collisions) can be used as the definition for authenticators. We have done the proofs in the remainder of this paper for the original definitions as well as the following, simpler, *auth* function.

```
auth :: Index → Digest → [Digest] → Digest
auth j datumDig lvlDigs =
  hash (concat (encode j : datumDig : lvlDigs))
```

3.1 Advancement Proofs

As more data is appended to the log, a client must be able to verify that the maintainer did not rewrite history, that is, that the appended log entries are consistent with any previous log entries that the client has already verified. For this purpose, a maintainer constructs an advancement proof, which can be checked by a client that already knows and trusts the hash (authenticator) h_i for the ith log entry in order to verify that the hash h_j for some $j > i$ is consistent with extending the log from position i to position j.

Depending on the use case, the advancement proof could be sent in response to an explicit request, or the sender could know what the recipient already knows and therefore what advancement proof it requires. Next, we examine how the hashing strategy described above enables efficient construction and verification of these advancement proofs.

An advancement proof comprises a Merkle path [8, 10] in the skiplog. For example, consider the skiplog in Figure 1(a). Suppose Alice knows and trusts a value (call it a_7) for the cumulative hash at position 7, but has not yet verified that the log advanced from index 7 to 12. To verify that the advancement from 7 to 12 is consistent with the log she already trusts up to index 7, Alice needs enough information in addition to the hashes she already trusts to compute a value for h_{12}. Figure 1(b) illustrates this information: circles represent information Alice needs, squares represents information she

already has, and the bold links show the path from 12 to 7; superfluous links have been erased for clarity.

If Alice can obtain h_4, h_6, h_{10}, h_{11} and the hashes of the data in positions 12 and 8 (d_{12} and d_8), then she can compute a value for h_{12} using a_7 (the value for index 7 that she already trusts), along with the genesis hash $h_\star$. Note that Alice's possession of a value for the hash at index 7 does *not* imply that she also knows h_4 or h_6: a_7 is computed by hashing a value that depends on the data at index 7 and h_6, so Alice would have to invert the hash function to get h_6 from h_7.

$$a_{12} = auth\ 12\ d_{12}\ [\, h_{11}, h_{10}, auth\ 8\ d_8\ [\, a_7, h_6, h_4, h_\star\,]\,]$$

In some contexts (such as in our motivating use case), Alice may already know the expected value for h_{12} (e.g., because consensus has been reached on it). In this case, she can compare her computed value a_{12} to h_{12} and confirm that the advancement from index 7 to 12 is consistent with the log that led to h_{12}. We encode the information Alice needs in the *AdvProof* type.

```
data AdvProof = Done
              | Hop Index Digest [Digest] [Digest] AdvProof
```

The structure of an advancement proof mimics a traversal through the skip pointers in a skip list. For example, an advancement proof from index 7 to index 12 sent to Alice can be represented by the following value p of type *AdvProof*:

$$Hop\ 12\ d_{12}\ [\, h_{11}, h_{10}\,]\ [\,]\ (Hop\ 8\ d_8\ [\,]\ [\, h_6, h_4, h_\star\,]\ Done)$$

Each *Hop* carries all the information needed to build an authenticator for the hop's source index using *auth*. For example, p's first hop (from index 12) contains the datum hash d_{12} and information sufficient to determine the authenticators for each dependency of index 12, thus enabling the verifier to compute a_{12} using *auth*, as shown above. The advancement proof contains two lists of authenticators for each hop: one for dependencies of the hop's source that are *after* the hop's target index, and one for those that are *before* it. In the example, the first list of the hop from 12 contains authenticators for 11 and 10, which are after 8 (the hop's target index), and there are no dependencies before 8 (note that the authenticator for the genesis index $\star$ is not needed because it is known by all in advance).

Finally the hop provides another advancement proof from the hop's target to the final target of the advancement proof (7 in this case), enabling recursive computation of the authenticator for the hop's target (8 in the example).

This structure enables us to *rebuild* an authenticator from an advancement proof by recursively computing the authenticator for the hop's target, and then using *auth* to combine this with the supplied authenticators for the other dependencies. When the recursion reaches the base case *Done*, the authenticator value provided to the *rebuild* function is used (⧺ is list concatenation).

```
rebuild :: AdvProof → Digest → Digest
rebuild Done                    d = d
rebuild (Hop j datDig af bf prf) d =
    auth j datDig (af ⧺ [rebuild prf d] ⧺ bf)
```

One can verify that $rebuild\ p\ a_7$ constructs the correct hash. Note that $rebuild\ Done\ a_7$ provides the hash Alice already trusts for index 7.

Building an advancement proof is done by traversing the dependency graph of the log. The smallest advancement proof is obtained by traversing from position j to $i < j$ via the highest level hop l such that $j - 2^{l-1} \geqslant i$; this is facilitated by the *singleHopLevel* function.

```
singleHopLevel :: Index → Index → Level
singleHopLevel from to =
  min (1 + floor (logBase 2 (from − to))) (maxLvl from)
```

The *mkadv* function uses this to traverse the dependency graph of a log and construct an advancement from j to i:

```
mkadv :: Index → Index → Log a → AdvProof
mkadv i j log =
  if i ≡ j
  then Done
  else let Auth dat datDig = lookup j log
           sh  = [lookupHash log (j − 2^(l−1)) | l ← [1 .. maxLvl j]]
           hop = singleHopLevel j i
           af  = take (hop − 1) sh
           bf  = drop hop sh
       in Hop j datDig af bf (mkadv i (j − 2^(hop−1)) log)
```

Advancement proofs can also be used to construct membership proofs: a maintainer can prove to a client that data d_i is at position i, if the client trusts the cumulative hash at position j, for $j > i$. The maintainer sends an advancement proof from i to j, along with a list of the authenticators of the dependencies of i; the client can then compute and confirm the authenticator at i.

```
type MembershipProof = (AdvProof, [Digest])

isMemberAt :: (Hashable a)
           ⇒ MembershipProof → Index → a → Digest
           → Bool
isMemberAt (adv, ss) i a trustedRoot =
  rebuild adv (auth i (hash a) ss) ≡ trustedRoot
```

Constructing values of type *MembershipProof* is trivial: We construct an advancement proof and add the necessary authenticators to it. Membership proofs can also be used to establish relative order between elements. Element a is at an index smaller than element b iff there is a membership proof from the index of b to the index of a.

3.1.1 Well-Formed and Normalized Proofs. It is easy to construct values of type *AdvProof* that are not valid advancement proofs. For example, here are two *non-well-formed* advancements from index 4 to index 12:

$w_1 = Hop\ 12\ d_{12}\ [h_{11}, h_{10}]\ [\,]\ (Hop\ 8\ d_8\ [h_7]\ [h_\star]\ Done)$

$w_2 = Hop\ 12\ d_{12}\ [h_{11}, h_{10}]\ [\,]\ (Hop\ 6\ d_6\ [h_5]\ [\,]\ Done)$

In w_1, the *Hop* 8 step is missing an authenticator. Recall that the authenticator for position i depends on $maxLvl\ i$ authenticators. We are supposed to compute one authenticator recursively, which leaves the other $maxLvl\ i - 1$ of them to be specified by the advancement proof. Index 8 has level 4, so the total number of authenticators in the *before* and *after* lists should be exactly 3. For w_2, the first hop should be at level 2, from index 12 to $12 - 2^2 \equiv 8$, but the proof declares that the hop lands at index 6.

Consequently, we must identify what it means for a proof to be well formed—that is, it has the correct number of authenticators and takes the correct hop at every step.

$$
\begin{aligned}
&wellformed :: AdvProof \to Index \to Index \to Bool\\
&wellformed\ Done\ j\ i \qquad\qquad\quad = (j \equiv i)\\
&wellformed\ (Hop\ k\ \_\ af\ bf\ r)\ j\ i = (k \equiv j)\\
&\qquad\qquad\qquad\qquad \wedge (length\ (af +\!\!+ bf) + 1 \equiv maxLvl\ j)\\
&\qquad\qquad\qquad\qquad \wedge (wellformed\ r\ (j - 2^{length\ af})\ i)
\end{aligned}
$$

Well-formedness is a purely structural constraint that is trivial to check in practice. In our formal development, we enforce the *wellformed* invariant at the type level, which precludes construction of non-well-formed proofs.

Now, consider the following *AdvProof* from 4 to 12:

$$
\begin{aligned}
w_3 = Hop\ 12\ d_{12}\ [h_{11}]\ [h_8]\ (Hop\ 10\ d_{10}\ [h_9]\ [\,]\\
\quad (Hop\ 8\ d_8\ [h_7]\ [h_4, h_\star]\ (Hop\ 6\ d_6\ [h_5]\ [\,]\ Done)))
\end{aligned}
$$

Although w_3 is a well-formed proof, it takes more hops than necessary. A *normalized* proof takes the hop determined by *singleHopLevel* at each step, resulting in the shortest possible advancement proof, thus reducing bandwidth and computation requirements for sending and verifying them. Note that our results hold for advancement proofs formed by composition, which are not always normalized.

3.1.2 Size of Advancement Proofs. A normalized advancement proof between indexes i and $j > i$ has at most $2\lceil \log_2(1 + j - i) \rceil$ hops (we go up hop levels to get from j to the maximum-length hop that does not overshoot i, and each hop makes twice as much progress as the previous; similarly, we then go down from that hop to reach i, with each hop progressing at least half the way to i). Each hop references at most $\lceil \log_2 j \rceil$ digests. These observations yield a bound of at most $2\lceil \log_2(1 + j - i) \rceil \lceil \log_2 j \rceil$ digests.

This bound is conservative: there may be fewer hops, fewer digests per hop (and the maximum per hop is smaller for smaller indexes), and each digest is included only once, even if it is referenced by multiple hops in an advancement proof (see the *View* type in Section 4.1).

Proving a tighter bound is left as future work. However, using our Haskell specification to examine all normalized advancement proofs between indexes less than 1,000 yields several observations. The advancement proof from $i = 1$ to $j = 991$ is both the longest (visiting 17 indexes) and the largest (85 digests). About 40 digests are included on average, which is about 1/4 of the number indicated by the conservative bound. Avoiding duplicate digests (Section 4.1) saves only about 3 digests on average, but results in a worthwhile simplification to the algorithm and proofs.

4 Verifying Key Properties Using Agda

The original AAOSL work [6] presents manual proofs of several properties, showing that they can be violated only by an adversary that finds a collision for the underlying cryptographic hash function, which is assumed to be infeasible for a computationally bounded adversary. We have constructed an Agda proof of the main property specified by Maniatis and Baker, called *Evolutionary collision resistance of AAOSL membership proofs* (Theorem 3). Following the original naming, we call this property EVO-CR.

The essence of the EVO-CR property is that, if two clients have advanced their logs to a point with the same digest, then it is infeasible for a computationally bounded adversary to convince them to accept different authenticators for the same earlier log position, even if the clients advanced their logs via different advancement proofs. Furthermore, if the hop relation contains a hop between every index (except the initial index) and its predecessor—as with the hop relation defined in Section 3—then there is a "degenerate" advancement proof that visits every index between 0 and j, for any j. In this case, AGREEONCOMMON implies that a computationally bounded adversary cannot convince a client to accept a membership proof for index i that is inconsistent with a given log at index i if the client has rebuilt the membership proof to the same authenticator as that log at index $j \geqslant i$.

Our proof of EVO-CR is based on three applications of a key property that we prove first, called AGREEONCOMMON. This property states that, for any two advancement proofs into a new index j, if rebuilding both proofs yields the same hash, then either these proofs agree on the authenticator for every index that they both visit, or there is a hash collision.

The overall proof is divided into two parts: an abstract model of a *class* of authenticated skip lists, and a concrete instantiation thereof, which we prove meets the necessary requirements. The abstract model assumes a *DepRel* (defined below). This type implies a dependency relation satisfying several properties that are introduced below. Our concrete skiplog (described in Section 3) is defined by instantiating the abstract model with a *DepRel* that implies the dependency

relation described in Section 3. The same could be achieved for a wide variety of such dependency relations.

The *DepRel* record defines the class of skiplogs for which we prove AgreeOnCommon and then evo-cr.

```
record DepRel : Set where
  field
    maxlvl    :  ℕ → ℕ
    maxlvl-z  :  maxlvl 0 ≡ 0
    maxlvl-s  :  (m : ℕ) → 0 < maxlvl (suc m)
  HopFrom     :  ℕ → Set
  HopFrom     =  Fin ∘ maxlvl
  field
    hop-tgt   :  {m : ℕ} → HopFrom m → ℕ
    hop-inj   :  {m : ℕ} {h₁ h₂ : HopFrom m}
              →  hop-tgt h₁ ≡ hop-tgt h₂ → h₁ ≡ h₂
    hop≺      :  {m : ℕ} (h : HopFrom m) → hop-tgt h < m
    hop-no-cross : {j₁ j₂ : ℕ} {h₁ : HopFrom j₁}
                                {h₂ : HopFrom j₂}
              →  hop-tgt h₂ < hop-tgt h₁ → hop-tgt h₁ < j₂
              →  j₁ ⩽ j₂
```

The *DepRel* definition requires a *maxlvl* function, which should return the number of hops from a given index, and a proof that the level of 0 is 0, i.e., there are *no* hops out of the initial index. For every index i there are *maxlvl* i hops, encoded by the *HopFrom* datatype, and these hops have targets (*hop-tgt*). It is important that *hop-tgt* is injective (*hop-inj*), but also that taking a hop *progresses*, that is, the target of a hop is always smaller than the source (*hop*$\prec$). It is also important that hops never cross over each other, which is ensured by *hop-no-cross*. This property is illustrated in Figure 2; we examine it in more detail when we describe how we ensure our concrete hop relation respects it in Section 4.4.

Constructing an inhabitant of *DepRel* with *maxlvl* as defined in Section 3 and *hop-tgt* $\{j\}$ h defined as $j - 2^h$ is mostly straightforward. However, although one can marvel at a geometric proof of the *hop-no-cross* property for this hop relation on a napkin, constructing a precise, machine checked proof of this property is a substantial challenge. In the second part of our proof, presented in Section 4.4, we prove that we can instantiate the abstract model with the hop relation used by our particular implementation.

4.1 Abstract Model

Our abstract model in Agda requires an arbitrary value of type *DepRel* as a module parameter, so the properties in this section apply to any dependency relation implied by *DepRel*'s requirements. Next, we introduce base notions necessary for constructing an AAOSL.

To avoid the inconsistency issues discussed in Section 3.1.1, we represent an advancement proof using a well-formed *advancement path* that simply indicates which hops the proof takes, along with the datum hashes of the source of each hop.

```
data AdvPath : ℕ → ℕ → Set where
  Done  :  ∀ {i} → AdvPath i i
  Hop   :  ∀ {j i} → Hash → (h : HopFrom j)
        →  AdvPath (hop-tgt h) i
        →  AdvPath j i
```

A prover provides the authenticators associated with the indexes in an advancement path in a separate *view*, which we *model* as a function from log indexes to hashes:

```
View : Set
View = ℕ → Hash
```

This way, a single authenticator is provided for each index, regardless of how many times that index appears in the proof or is a dependency of an index that appears in the proof. (A practical Haskell implementation would represent a view as a partial map: *Data.Map Index Hash*; if the map does not include an authenticator for an index required by the advancement proof, then the advancement proof is rejected.)

The *rebuild* function in Agda operates over *View*s instead of receiving and returning a single hash. This is important because it enables us to *lookup* all rebuilt authenticators from a proof. For example, take a = *Hop* d_8 3 (*Hop* d_4 3 *Done*). Calling *rebuild* a t, for some view t, will return another view t', where we can lookup the newly rebuilt value for 8 with t' 8, and also check the recursively rebuilt hash for 4, used in the computation of t' 8, by calling t' 4.

```
rebuild : ∀ {i j} → AdvPath j i → View → View
rebuild Done                 view = view
rebuild (Hop {j = j} x h a) view =
  let view′ = rebuild a view
  in insert j (auth j x view′) view′
```

Here, *insert* k v f inserts a new key-value pair into a map. Note how we are computing the authenticator for j using the view that results from recursively rebuilding the proof.

Next we look at encoding *membership proofs*, which reuse the constructions we have just seen. Although similar, membership proofs and advancement proofs work in different "directions". An advancement proof is used to prove to someone who trusts the hash of index i that the skiplog can be extended to index $j > i$ in a way that is consistent with the log up to index i. Membership proofs, on the other hand, prove to a verifier who trusts the hash of $j > i$, that a given datum is at index i.

As described above, when rebuilding an *advancement proof* from i to j, a verifier already knows and trusts an authenticator for index i. In contrast, with *membership proofs*, the prover must provide sufficient additional information for the verifier to compute the authenticator for index i. The datum hash is provided explicitly as the second component of the membership proof. The third component ensures that we do

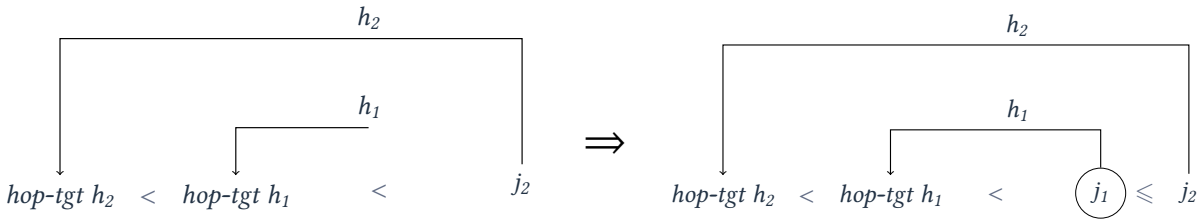


Figure 2. Graphical representation of *hop-no-cross*

not construct a *MbrPath* for index 0, which does not have associated data. (Agda uses × to express product types.)

```
MbrPath : ℕ → ℕ → Set
MbrPath j i = AdvPath j i × Hash × i ≢ 0

datum-hash : ∀ {j i} → MbrPath j i → Hash
datum-hash (_, dat, _) = dat

mbr-path : ∀ {j i} → MbrPath j i → AdvPath j i
mbr-path (p, _, _) = p
```

In addition to a *MbrPath*, the prover also provides a *View* that the verifier uses to rebuild the *MbrPath*:

```
rebuildMbr : ∀ {j i} → MbrPath j i → View → View
rebuildMbr {j} {i} mp t =
  rebuild (mbr-path mp) (insert i (auth i (datum-hash mp) t) t)
```

Unlike with advancement proofs, the provided *View* must also include authenticators for each dependency of index i that—together with the datum hash—enable the verifier to compute the authenticator for index i.

Reasoning about collision resistance. The proofs presented in the next sections establish desired properties modulo hash collisions. Hence, it is important to model the possibility that an adversary may find hash collisions. One option for doing this in a constructive setting is to carry around hash collisions evidenced in existentials [1]. We use the *HashBroke* datatype:

```
HashBroke : Set
HashBroke = ∃[x, y] (x ≢ y × hash x ≡ hash y)
```

This becomes necessary when reasoning about injectivity of authenticators, which is central to our proofs. If two advancement proofs $a_1 : AdvPath\ j\ i_1$ and $a_2 : AdvPath\ j\ i_2$ rebuild using views t_1 and t_2 respectively to the same authenticator at j (i.e., $rebuild\ a_1\ t_1\ j \equiv rebuild\ a_2\ t_2\ j$), then *unless there is a hash collision*, a_1 and a_2 provide the same datum hash for j and both rebuilds use the same authenticators for all dependencies of j.

This conclusion is reached via two injectivity lemmas applied to the evidence that the rebuilt views agree at j. The first establishes that (unless *HashBroke*), the datum hashes passed to *auth* to compute the authenticators for index j for the respective advancement proofs are equal:

```
auth-inj-1 : {j : ℕ} {h₁ h₂ : Hash} {t₁ t₂ : View}
  → j ≢ 0 → auth j h₁ t₁ ≡ auth j h₂ t₂
  → Either HashBroke (h₁ ≡ h₂)
```

The *auth-inj-1* lemma is says that, if the *datumDig* values passed to two invocations of *auth* are different, but they return the same hash, then we have a hash collision. Its proof is somewhat more involved, though, because *auth* uses concatenations and encoding of natural numbers into bytestrings. Therefore the proof requires reasoning about injectivity of encoding and of concatenation of fixed-size byte strings. We make a simplifying assumption that indexes are encoded into a fixed (but unspecified) number of bits. This assumption is reasonable in practice and not difficult to relax, but this would not add significant value to our proof.

Once we have *auth-inj-1* to determine that the datum hashes are equal (unless *HashBroke*), we define the other injectivity lemma to conclude that the lists of digests provided to the *lvldigs* arguments of the respective *auth* invocations are also equal:

```
auth-inj-2 : ∀ {j h t₁ t₂} → auth j h t₁ ≡ auth j h t₂
  → Either HashBroke (Agree t₁ t₂ (depsof j))
```

Where $Agree\ t_1\ t_2\ xs$ states that $t_1\ x \equiv t_2\ x$ for every $x \in xs$ and *depsof* returns the dependencies of a given index j by enumerating all hops from j and computing their target. In case j is zero, it has no dependencies.

```
depsof : ℕ → List ℕ
depsof 0       = [ ]
depsof (suc i) = map hop-tgt (finsUpTo (lvlof (suc i)))
```

We have proven that both the original definition of *auth* used by Maniatis and Baker [6] and the simpler variant described in Section 3 satisfy these two injectivity lemmas. In fact, any definition of *auth* that satisfies *auth-inj-1* and *auth-inj-2* could be used to construct an AAOSL that enjoys all the properties we will explore next.

4.2 Proving AgreeOnCommon

The AgreeOnCommon property states that, given two advancement proofs into index j, if both proofs rebuild to the same hash, then either these proofs agree on the authenticators of *every* index that they visit in common, or the

adversary found a hash collision. This is stated in Agda as follows:

```
AgreeOnCommon
  : ∀ {j i₁ i₂} {t₁ t₂ : View}
  → {a₁ : AdvPath j i₁} {a₂ : AdvPath j i₂}
  → rebuild a₁ t₁ j ≡ rebuild a₂ t₂ j
  → {i : ℕ} → i ϵ_AP a₁ → i ϵ_AP a₂
  → Either HashBroke (rebuild a₁ t₁ i ≡ rebuild a₂ t₂ i)
```

Here, $\_\epsilon_{AP}\_$ encodes the relation of visited indexes of a given advancement proof. A value of type $k\ \epsilon_{AP}\ a$ exists iff index k is visited by advancement proof a:

```
data _ϵ_AP_ (k : ℕ) : {j i : ℕ} → AdvPath j i → Set where
  hereTgtDone : k ϵ_AP (Done {k})
  hereTgtHop  : {i : ℕ} {d : Hash} {h : HopFrom k}
                {a : AdvPath (hop-tgt h) i}
                → k ϵ_AP (Hop d h a)
  step        : {i j : ℕ} {d : Hash} {h : HopFrom j}
                {a : AdvPath (hop-tgt h) i}
                → k ≢ j → k ϵ_AP a
                → k ϵ_AP (Hop d h a)
```

The proof of AgreeOnCommon follows by induction on the commonly visited indexes of the advancement proofs under scrutiny. We present the proof in two parts. First, we explain how we encode this induction principle in a datatype, *AOC*, and why it provides sufficient information to prove AgreeOnCommon. Later we prove that we can always construct a value of type *AOC* for the advancement proofs expected by AgreeOnCommon.

Using the *AOC* data type to prove AgreeOnCommon. In this section we focus on how *AOC* captures enough information to conclude that two advancement proofs rebuild to the same hash at *every* index they both visit. Intuitively, a value of type $AOC\ t_1\ t_2\ a_1\ a_2$ represents a list of all indexes visited by both a_1 and a_2, along with sufficient evidence to prove that the respective views obtained by rebuilding a_1 using t_1 and rebuilding a_2 using t_2 agree at each of these indexes. This can be seen in the type of *aoc-correct*, below.

```
aoc-correct
  : ∀ {j i₁ i₂} {a₁ : AdvPath j i₁} {a₂ : AdvPath j i₂}
  → {t₁ t₂ : View}
  → AOC t₁ t₂ a₁ a₂
  → {i : ℕ} → i ϵ_AP a₁ → i ϵ_AP a₂
  → Either HashBroke (rebuild a₁ t₁ i ≡ rebuild a₂ t₂ i)
```

The proof of *aoc-correct* is a long but straightforward induction on *AOC* and $i\ \epsilon_{AP}\ a_1$ and $i\ \epsilon_{AP}\ a_2$. The more intellectual step lies in defining *AOC* to have the right structure to make the rest of the proof straightforward. Intuitively, *AOC* is like a *zip* over advancement proofs: it puts in evidence the commonly visited indexes. Its type signature reveals that it is a relation between *AdvPath*s from a common index j:

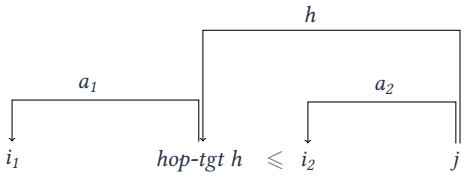
```
data AOC (t₁ t₂ : View)
  : ∀ {i₁ i₂ j} → AdvPath j i₁ → AdvPath j i₂ → Set where
```

Each of *AOC*'s constructors represents one possible situation with advancement proofs that share a common index. There are three base cases that handle pairs of advancement proofs a_1 and a_2 that have one or two common indexes: (i) both proofs are *Done*, (ii) a_1 hops over a_2, or (iii) a_2 hops over a_1. These are captured by the following *AOC* constructors:

```
PDoneDone : {i : ℕ} → t₁ i ≡ t₂ i
          → AOC t₁ t₂ {i} {i} Done Done
POverR    : ∀ {i₁ i₂ j} {d : Hash} {h : HopFrom j}
          → (a₁ : AdvPath (hop-tgt h) i₁) (a₂ : AdvPath j i₂)
          → hop-tgt h ⩽ i₂
          → rebuild (Hop d h a₁) t₁ j ≡ rebuild a₂ t₂ j
          → AOC t₁ t₂ (Hop d h a₁) a₂
POverL :  -- symmetric
```

The first constructor *PDoneDone* represents pairs of advancement proofs that trivially agree on their only common index because both proofs are *Done* and the two views agree at that index.

In the second constructor, *POverR*, the left advancement path is not *Done*: it takes a hop $Hop\ d\ h\ a_1$. Moreover, h hops *over* a_2 (note that the constructor requires evidence that $hop\text{-}tgt\ h \leqslant i_2$). The diagram below illustrates this case.



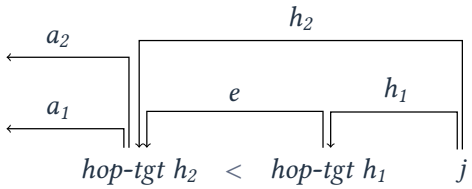
POverR requires evidence that $rebuild\ (Hop\ d\ h\ a_1)\ t_1$ agrees with $rebuild\ a_2\ t_2$ on j, which is sufficient to ensure that both proofs rebuild to the same hash at j. It also requires evidence that $hop\text{-}tgt\ h \leqslant i_2$, which implies that the advancement proofs can have at most one more commonly visited index besides j, which is i_2 when $hop\text{-}tgt\ h \equiv i_2$. In this case, because i_2 would then be a dependency of j (evidenced by the implicit hop h provided to *POverR*), applying *auth-inj-1* and *auth-inj-2* to the hypothesis that $Hop\ d\ h\ a_1$ and a_2 rebuild the same authenticator at j, yields $Agree\ t_1\ t_2\ (depsof\ j)$, implying that the digests for i_2 in the two advancement proofs are equal (again, unless *HashBroke*). This implies that the *View*s resulting from rebuilding the two advancement proofs used the same authenticators for index i_2; some simple lemmas ensure that rebuilding hops to higher indexes do not modify the views at i_2, so the rebuilt views also agree at i_2, as required.

POverL is symmetric to *POverR*, capturing the case in which the right advancement proof hops over the entire left advancement proof.

Next we examine the three inductive cases of *AOC*. The simplest is *PCong*, which is used when both proofs take the same hop. *PCong* requires evidence *AOC* $t_1\ t_2\ a_1\ a_2$ that the rebuilt views agree at all existing common indexes, as well as sufficient evidence to conclude that *rebuild* (*Hop* $d\ h\ a_1$) $t_1\ j$ is equal to *rebuild* (*Hop* $d\ h\ a_2$) $t_2\ j$.

```
PCong : ∀ {i₁ i₂ j} {d} {h : HopFrom j}
      → (a₁ : AdvPath (hop-tgt h) i₁)
      → (a₂ : AdvPath (hop-tgt h) i₂)
      → Agree (rebuild a₁ t₁) (rebuild a₂ t₂) (depsof j)
      → AOC t₁ t₂          a₁          a₂
      → AOC t₁ t₂ (Hop d h a₁) (Hop d h a₂)
```

Finally, the case in which a_2 is extended to index j by a hop h_2 and a_1 is extended by a *different* path to index j is handled by the *PMeetR* constructor; *PMeetL* being its symmetric counterpart. This situation is captured in the diagram below.



The idea here is that we can look at the advancement proof that takes the smaller hop as a composition of two advancement proofs making evident the next common index.

```
PMeetR : ∀ {j i₁ i₂ d} {h₁ h₂ : HopFrom j}
       → (e : AdvPath (hop-tgt h₁) (hop-tgt h₂))
       → (a₁ : AdvPath (hop-tgt h₂) i₁)
       → (a₂ : AdvPath (hop-tgt h₂) i₂)
       → hop-tgt h₂ < hop-tgt h₁
       → Agree (rebuild (e ⊕ a₁) t₁) (rebuild a₂ t₂) (depsof j)
       → AOC t₁ t₂ a₁ a₂
       → AOC t₁ t₂ (Hop d h₁ (e ⊕ a₁)) (Hop d h₂ a₂)
PMeetL :  -- symmetric
```

Here, _⊕_ composes advancement proofs.

```
_⊕_ : ∀ {i j k} → AdvPath j k → AdvPath k i → AdvPath j i
```

PMeetR and *PMeetL* are similar to *PCong* in that they all require an *AOC* $t_1\ t_2\ a_1\ a_2$, which provides evidence that the smaller advancement proofs agree at each of their common indexes. They also all require sufficient evidence to prove that the authenticator computed for the new common index j is the same following either of the new advancement proofs to index j.

One important observation is that rebuilding an advancement proof never interferes with previous indexes. That is, rebuilding a path *e*:*AdvPath* $j_1\ j_2$ on a view that was obtained from rebuilding a : *AdvPath* $j_2\ i$ on a view t will not change any authenticator already computed for any $k \leqslant i$. This is witnessed by the *rebuild-⊕*property below, which guarantees that the inductive hypothesis provided by *AOC* $t_1\ t_2\ a_1\ a_2$ in *PMeetR* is not falsified by rebuilding $(e \oplus a_1)$ on top of t_1.

```
rebuild-⊕: ∀ {j₁ j₂ i k t} (e : AdvPath j₁ j₂) (a₁ : AdvPath j₂ i)
         → k ∈ₐₚ a₁
         → rebuild (e ⊕ a₁) t k ≡ rebuild a₁ t k
```

This property is proved by a simple induction using the fact that computing authenticators for indexes higher than j_2 does not modify the provided view at j_2 or any lower index because $l\ \epsilon_{\text{AP}}\ a_1$ and a_1 is an *AdvPath* $j_2\ i$, implying that $i \leqslant j_2$).

Constructing an *AOC* inhabitant. To conclude the proof of AgreeOnCommon, we must construct an *AOC* $t_1\ t_2\ a_1\ a_2$ given *rebuild* $a_1\ t_1\ j \equiv$ *rebuild* $a_2\ t_2\ j$, where a_1 and a_2 are advancement proofs to index j. This is accomplished by the AOC lemma.

```
AOC : ∀ {i₁ i₂ j} (t₁ t₂ : View) (a₁ : AdvPath j i₁) (a₂ : AdvPath j i₂)
    → i₁ ⩽ i₂ -- w.l.o.g
    → rebuild a₁ t₁ j ≡ rebuild a₂ t₂ j
    → Either HashBroke (AOC t₁ t₂ a₁ a₂)
```

First, we pattern match on a_1 and a_2 and compare the *hop-tgt* (if any) of the hops taken. This enables us to understand which constructor of *AOC* must be used. In most cases we can easily extract the evidence needed for the relevant *AOC* constructor via case analysis on the structure of the two proofs and the injectivity lemmas discussed above, applied to the hypothesis that *rebuild* $a_1\ t_1\ j \equiv$ *rebuild* $a_2\ t_2\ j$.

However, we face an additional challenge when the two advancement proofs take different hops, but neither hop passes the other's entire proof. These cases require the *PMeetR* or *PMeetL* constructor, and we must *split* the proof that took the shorter hop into two composable pieces in order to obtain an advancement proof between the targets of the hops taken by the respective advancement proofs. This is needed for the *e* argument to the constructor. Because of the *hop-no-cross* requirement on *DepRel* we can always perform this *split* as evidenced by the lemma below. No hop can cross from an index strictly between *hop-tgt* h_2 and *hop-tgt* h_1 to an index lower than or equal to *hop-tgt* h_1 without passing through *hop-tgt* h_1 first.

```
split : ∀ {j₁ j₂ i} {h₁ : HopFrom j₁} {h₂ : HopFrom j₂}
      → j₂ ⩽ j₁
      → hop-tgt h₁ < hop-tgt h₂
      → i ⩽ hop-tgt h₁
      → (a : AdvPath (hop-tgt h₂) i)
      → ∃[x : AdvPath (hop-tgt h₂) (hop-tgt h₁), y]
          (a ≡ x ⊕ y)
```

4.3 Proving EVO-CR

With AGREEONCOMMON out of the way, we move to the next property. The EVO-CR property [6] states that a computationally bounded adversary cannot produce two advancement proofs a_1 and a_2 that rebuild to the same authenticator at some index j, and also provide two membership proofs from s_1 to tgt and s_2 to tgt that prove a conflicting claim at tgt, given that a_1 visits s_1 and tgt and a_2 visits s_2 and tgt. This is formalized by the type of EVO-CR and illustrated in Figure 3.

```
EVO-CR : ∀ {j i₁ i₂} {t₁ t₂ : View}
       → (a₁ : AdvPath j i₁) (a₂ : AdvPath j i₂)
       → rebuild a₁ t₁ j ≡ rebuild a₂ t₂ j
       → ∀ {s₁ s₂ tgt} {u₁ u₂ : View}
       → (m₁ : MbrPath s₁ tgt) (m₂ : MbrPath s₂ tgt)
       → s₁ ϵ_AP a₁ → s₂ ϵ_AP a₂
       → tgt ϵ_AP a₁ → tgt ϵ_AP a₂
       → rebuildMbr m₁ u₁ s₁ ≡ rebuild a₁ t₁ s₁
       → rebuildMbr m₂ u₂ s₂ ≡ rebuild a₂ t₂ s₂
       → Either HashBroke (datum-hash m₁ ≡ datum-hash m₂)
```

The first step in proving EVO-CR is to split a_1 into $a_{13} \oplus a_{12} \oplus a_{11}$ and similarly for a_2, as depicted in Figure 3. Now, we rely on AGREEONCOMMON three times:

1. First, we notice that $a_{12} \oplus a_{11}$ and m_1 arrive and agree at s_1 and both visit tgt, therefore AGREEONCOMMON implies that they agree on tgt. This gives us

   $$rebuild\ (a_{12} \oplus a_{11})\ t_1\ tgt \equiv rebuildMbr\ m_1\ u_1\ tgt$$

2. Analogously to (i), we look at $a_{22} \oplus a_{21}$ and m_2. By AGREEONCOMMON, this implies that

   $$rebuild\ (a_{22} \oplus a_{21})\ t_2\ tgt \equiv rebuildMbr\ m_2\ u_2\ tgt$$

3. Finally, we note that a_1 and a_2 arrive and agree at j and have tgt as a commonly visited index. Applying AGREEONCOMMON for the last time with *rebuild-⊕* gives us:

   $$rebuild\ (a_{12} \oplus a_{11})\ t_1\ tgt \equiv rebuild\ (a_{22} \oplus a_{21})\ t_2\ tgt$$

We now compose the equalities obtained in (1), (2) and (3) and conclude that $rebuildMbr\ m_1\ u_1\ tgt$ must be equal to $rebuildMbr\ m_2\ u_2\ tgt$. Recall that $rebuildMbr\ m_1\ u_1\ tgt$ is defined by:

```
rebuild (mbr-path m₁)
        (insert tgt (auth tgt (datum-hash m₁) u₁) u₁)
        tgt
```

Because tgt is the last index visited by $mbr\text{-}path\ m_1$, the view returned by rebuild does not overwrite the value for tgt that we inserted in the snippet above, $auth\ tgt\ (datum\text{-}hash\ m_1)\ u_1$. Hence, we have that:

$$rebuildMbr\ m_1\ u_1\ tgt \equiv auth\ tgt\ (datum\text{-}hash\ m_1)\ u_1$$

Applying the same reasoning for $rebuildMbr\ m_2\ u_2\ tgt$, we can conclude that $auth\ tgt\ d_1\ u_1$ must coincide with $auth\ tgt\ d_2\ u_2$, for $d_i = datum\text{-}hash\ m_i$. Finally, we use *auth-inj-1* to conclude $datum\text{-}hash\ m_1 \equiv datum\text{-}hash\ m_2$ and finalize our proof.

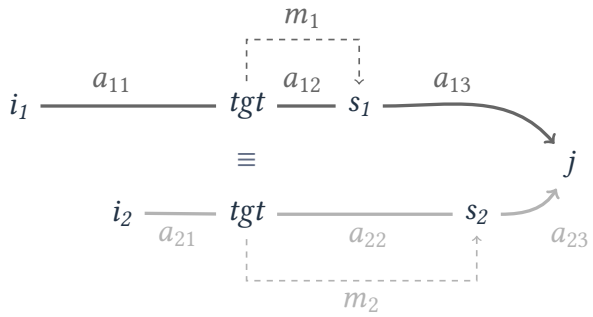


Figure 3. Graphical representation of EVO-CR. Thick lines represent the advancement paths, dashed lines represent the membership proofs.

This concludes the abstract properties. We have shown how any value of *DepRel* will yield a skiplog that enjoys AGREEONCOMMON and EVO-CR. The next step is proving that we can inhabit the *DepRel* datatype with the skiplog described in Section 3.

4.4 Instantiating the Abstract Model

Having proved that any instantiation of *DepRel* enjoys the AGREEONCOMMON and EVO-CR properties, we now instantiate *DepRel* with the hop relation defined in Section 3: we define *maxlvl* to be equivalent to the definition presented there (see below) and define hop targets as follows.

```
hop-tgt : {j : ℕ} → HopFrom j → ℕ
hop-tgt {j} h = j ∸ (2^toℕ h)
```

We use Agda's $\dot{-}$ operator, which truncates negative subtraction results to zero. Reasoning is simplified by a lemma proving that hop targets never "overshoot" zero.

In this section, we explain how we provide the remaining properties required to prove that the hop relation arising from the definitions of *maxlvl* and *hop-tgt* satisfies the requirements of *DepRel*. Most of them are straightforward. However, proving that this relation satisfies *hop-no-cross* is quite challenging. The first step towards a manageable proof is to define suitable induction principles over the hops we seek to analyze.

In our case, we want to prove a number of lemmas over the structure that arises from establishing $j - 2^l$ as a dependency of j, for l less than the largest power of two that divides j. Because $j - 2^{l+1} = (j - 2^l) - 2^l$, we can form hops at level $l + 1$ by composing adjacent hops at level l.

This structure is encoded in a simple Agda datatype:

```
data H : ℕ → ℕ → ℕ → Set where
  hz : ∀ x → H 0 (suc x) x
  hs : ∀ {l x y z}
     → H l x y → H l y z
     → suc l < maxLvl x
     → H (suc l) x z
```

A value h of type $H\ l\ j\ i$ proves the existence of a hop at level l connecting j and i. The constraint that $suc\ l < maxLvl\ x$ ensures that hops are created from index x only for levels up to $maxLvl\ x$. Without this constraint, for example, a hop from index 6 to index 2 could be constructed, which would cross the hop from 0 to 4. As neither of these hops is nested within the other, this would violate the *hop-no-cross* property, thus preventing a proof that our hop relation inhabits *DepRel*.

Another aspect that complicated our earlier direct proof efforts was the difficulty of dealing inductively with the original recursive definition of the *maxLvl* function, defined in Agda as:

```
maxLvl : ℕ → ℕ
maxLvl 0 = 0
maxLvl (suc n) with 2|?(suc n)
...|no _ = 1
...|yes e = suc (maxLvl (quotient e))
```

We tamed this complexity by observing that every non-zero natural number can be uniquely represented as the product of a power of two and an odd number, and then using a non-inductive definition of *maxLvl* that is isomorphic to our original definition. This is known as a *view* type in the literature [7].

```
data EvenOdd : ℕ → Set where
  zero : EvenOdd zero
  nz   : ∀ {n} l d → Odd d → n ≡ 2^l * d → EvenOdd n

to : (n : ℕ) → EvenOdd n
```

This enables us to write a non-inductive version of *maxLvl* that simply extracts this largest power of two.

```
maxLvl′ : ∀ {n} → EvenOdd n → ℕ
maxLvl′ zero        = zero
maxLvl′ (nz l _ _ _) = suc l
```

The definition of $maxLvl'$ is provably equivalent to the recursive version *maxLvl* used in the specification. The proof is an easy induction on n, and the lemma is used frequently throughout the proofs in our model and has been invaluable in helping us complete the proof.

```
maxLvl≡maxLvl : ∀ n → maxLvl n ≡ maxLvl′ (to n)
```

Next, we must be able to inhabit the type H, witnessing the existence of hops from a given index j into $j - 2^l$, as long as l is less than $maxLvl\ j$. This proves that the datatype is inhabited and meets the criteria we expect: i.e., it connects j and its dependencies one power of two away.

```
h-correct : ∀ j l → l < maxLvl j → H l j (j - 2^l)
```

Conversely, given $h : H\ l\ j\ i$, we can prove that i is a 2^l away from j, which shows that the H datatype encodes exactly the required structure.

```
h-univ_i : ∀ {l i j} → H l j i → i ≡ j - 2^l
```

A central lemma used in the proof that our hop relation satisfies the *hop-no-cross* property is that all the indexes that a hop skips over have a level lower than the level of the hop. In other words, if a hop from j to i at level l hops over index k, then $maxLvl\ k$ is at most l.

```
h-lvl-mid : ∀ {l j i} k → H l j i → i < k → k < j → maxLvl k ⩽ l
```

Next, we discuss how we prove that our hop relation satisfies the *hop-no-cross* property required by *DepRel*. Recalling the definition of this property, as illustrated in Figure 2, it would suffice to prove the following property:

```
nocross′ : ∀ {l₁ i₁ j₁ l₂ i₂ j₂} (h₁ : H l₁ j₁ i₁) (h₂ : H l₂ j₂ i₂)
         → i₂ < i₁ → Either (j₂ ⩽ i₁) (j₁ ⩽ j₂)
```

This property captures the intuition of hops *not crossing*: given any two hops that do not share an index (w.l.o.g. $i_2 < i_1$), either they do not overlap or one is contained within the other. (Note that two hops that share an index do not cross.) Our earlier attempts to prove *nocross′* directly became unmanageable due to the combined effects of dealing with arithmetic nuances, seemingly minor details such as subtraction in Agda truncating to zero to ensure the result is always a natural, and the fact that recursive calls do not directly inform us about the relationship between l_1 and l_2.

Therefore, we instead prove *hopcross* via a stronger property and an auxiliary lemma:

```
nocross : ∀ {l₁ i₁ j₁ l₂ i₂ j₂} (h₁ : H l₁ j₁ i₁) (h₂ : H l₂ j₂ i₂)
        → i₂ < i₁ → Either (j₂ ⩽ i₁) (h₁ ⊆ h₂)
⊆-src-⩽ : ∀ {l₁ i₁ j₁ l₂ i₂ j₂} (h₁ : H l₁ j₁ i₁) (h₂ : H l₂ j₂ i₂)
        → h₁ ⊆ h₂ → j₁ ⩽ j₂
```

The auxiliary lemma is proved via a straightforward induction on hops, using the following definition of $\subseteq$.

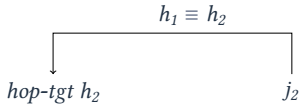
```
data _⊆_ : ∀ {l₁ i₁ j₁ l₂ i₂ j₂} → H l₁ j₁ i₁ → H l₂ j₂ i₂ → Set
  where
    here  : ∀ {l i j} (h : H l j i) → h ⊆ h
    left  : ∀ {l₁ i₁ j₁ l₂ i₂ w j₂} (h : H l₁ j₁ i₁)
              (w₀ : H l₂ j₂ w) (w₁ : H l₂ w i₂)
            → (p : suc l₂ < maxLvl j₂)
            → h ⊆ w₀ → h ⊆ (hs w₀ w₁ p)
    right : ... -- analogous to left
```

This definition establishes that every hop contains itself, and then recursively defines a hop h as being contained by

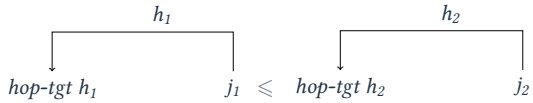
another if the latter is the composition of two adjacent hops, one of which contains h. However, this simplified description is too inclusive. Observe that the *left* and *right* constructors require constraints to enable the construction of legitimate hops using the *hs* constructor of H.

With these definitions established, we can describe the proof that our hop relation satisfies the *nocross* property, which proceeds by case analysis on hops. To facilitate this case analysis, we define an auxiliary relation that categorizes the relationship between two hops h_1 and h_2 into one of five possible situations, depending on the relative indexes of the hops' sources and targets:

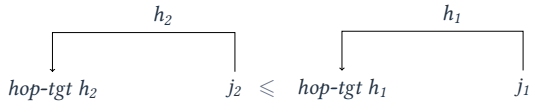
1. The hops are equal, i.e.:



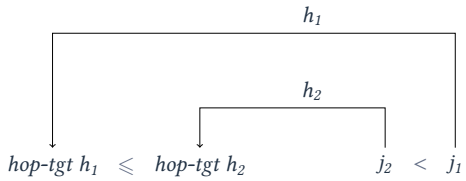
2. The hops are disjoint, with $\mathit{hop\text{-}src}\ h_1 \leqslant \mathit{hop\text{-}tgt}\ h_2$, i.e.:



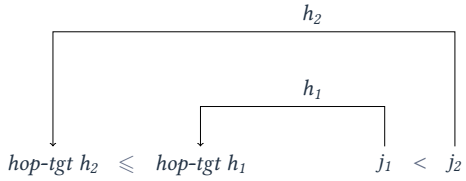
3. The hops are disjoint, with $\mathit{hop\text{-}src}\ h_2 \leqslant \mathit{hop\text{-}tgt}\ h_1$, i.e.:



4. The hops are nested but not equal, with h_2 being the shorter hop, i.e.:



5. The hops are nested but not equal, with h_1 being the shorter hop, i.e.:



It is interesting to note that the definition of the categorization mechanism (not shown) is mutually recursive with the definition of *nocross*: it uses *nocross* to distinguish some cases. Agda's termination checker confirms there is no circular reasoning here because it can prove that such recursive uses of *nocross* are on lower level hops, implying that the recursion must terminate.

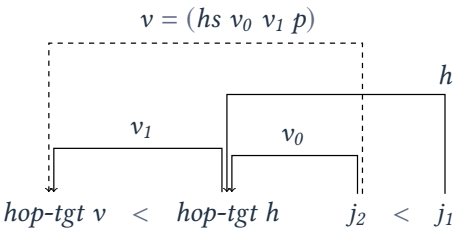


Figure 4. There exists no p with type $\mathit{suc}\ l < \mathit{maxLvl}\ j_2$ when in this situation. Hence, the composite hop $\mathit{hs}\ v_0\ v_1\ p$ cannot be built.

The proof of $\mathit{nocross}\ h\ v$ proceeds by induction on v. If $\mathit{hop\text{-}tgt}\ h \leqslant \mathit{hop\text{-}tgt}\ v$, then *nocross* holds vacuously. Otherwise, $\mathit{hop\text{-}tgt}\ v < \mathit{hop\text{-}tgt}\ h$, so if one hop is nested within the other, then it is h that is contained within v, the longer hop.

If v is a hop at level zero, we are done because any hop that hops over v must contain it. In case v is a composite hop $\mathit{hs}\ v_0\ v_1\ p$, we use the categorization mechanism to determine the relationship between h and v_0; in some cases we then use the categorization mechanism again to determine the relationship between h and v_1. It is straightforward in all cases except one to use information from the categorization mechanism to prove that:

- the case cannot arise; or
- $\mathit{hop\text{-}src}\ v \leqslant \mathit{hop\text{-}tgt}\ h$, implying that *nocross* holds via the "left" alternative; or
- $h \subseteq v_0$ or $h \subseteq v_1$, implying that $h \subseteq v$, so *nocross* holds via the "right" alternative.

The difficult case is illustrated in Figure 4. In this case, $\mathit{hop\text{-}tgt}\ h$ coincides with the point at which v_0 and v_1 meet, so there is no opportunity for the categorization mechanism to eliminate the case using a recursive instance of *nocross* against v_0 or v_1. It simply tells us that v_1 is disjoint from h and that h contains v_0.

The *h-lvl-mid* lemma is invaluable here: it proves that this case is impossible. Suppose v_0 and v_1 are hops at level l, so v is at level $\mathit{suc}\ l$. The contradiction arises from p, which is (supposedly) a proof that $\mathit{suc}\ l < \mathit{maxLvl}\ j_2$. Because h hops *over* j_2, *h-lvl-mid* implies that $\mathit{maxLvl}\ j_2$ is at most $\mathit{suc}\ l$.

Having proved these properties about our hop relation, we are ready to use it to instantiate the abstract model with our particular definition of *DepRel*. The properties required by *DepRel* other than *hop-no-cross* are straightforward to prove; some of them are shown below.

For *hop-no-cross*, first we define a simple convenience function to witness the existence of hops:

$$\mathit{getHop} : \forall\ \{j\}\ (h : \mathit{Fin}\ (\mathit{maxLvl}\ j)) \to H\ (\mathit{to}\mathbb{N}\ h)\ j\ (j - 2^{\mathit{to}\mathbb{N}\ h})$$
$$\mathit{getHop}\ \{j\}\ h = \mathit{h\text{-}correct}\ j\ (\mathit{to}\mathbb{N}\ h)\ (\mathit{to}\mathbb{N}{<}n\ h)$$

Then we prove the *hop-no-cross* property using *getHop* and *nocross*. We pattern match on the result of *nocross* and discharge the *Left* branch as impossible with simple arithmetic (not shown). The other branch carries a proof that one hop is entirely contained within the other, from which we extract a value of the desired type.

```
skiplog : DepRel
skiplog = record
  { maxlvl    = maxLvl
  ; maxlvl-z  = refl
  ; hop-tgt       = λ { m} h → m ∸ (2^toℕ h)
  ; hop-inj       = ... -- simple Agda exercises
  ; hop-≺         = ...
  ; hop-no-cross = λ { h₁ = h₁} { h₂} p1 p2 →
    case nocross (getHop h₁) (getHop h₂) of
      λ {Left imp → ...
        ; Right r  → ⊆-src-⩽ (getHop h₁) (getHop h₂) r}}
```

5 Related and Future Work

Pugh's original skip lists [16] supported insertion, necessitating the use of probabilistic "heights" (analogous to our deterministic *maxLvl* function) for elements added to the list. Authenticated skip lists proposed by Goodrich and Tamassia [3] similarly required probabilistic heights, and depended on the use of a commutative hash function. Maniatis and Baker [6] proposed an *append-only* authenticated skip list (AAOSL), eliminating the need for commutative hashing, and achieving a simpler structure better suited to representing tamper-evident logs such as blockchains. The concrete AAOSL achieved in Section 4.4 by instantiating our abstract model presented in Section 4.1 is essentially the same as [6], modulo our changes to eliminate the possibility of a prover providing different authenticators for the same index within an advancement proof. The "skipchains" used by Chaniac [12] use the same "hop structure" in the case of deterministic skipchains with $b = 2$; Chaniac also maintains "forward" links built when new elements are appended.

Authenticated data-structures have been studied in many different incarnations. Miller et al. [9] presents *lambda-auth*, a language used to write a large variety of authenticated data structures with ease. They provide pen-and-paper proofs that a prover can fool a verifier only if it finds a hash collision. Brun and Traytel [1] formalized *lambda-auth* in Isabelle/HOL, which corrected some issues in the original paper, strengthening the overall result and reinforcing the importance of formal verification. In authenticated data structures produced using *lambda-auth*, the hash of a value depends exclusively on its type structure. In our case, the hash of different indexes might have a different number of dependencies, even though each log entry carries only one recursive argument: its tail. It therefore does not seem possible to use *lambda-auth* to achieve encode the original AAOSL presentation [6], nor our formalization thereof.

Using trees instead of lists to enable efficient queries has also been done in the CCF framework [17], with a verified incremental Merkle tree implementation [15]. This enables logarithmic-time access to past transactions and enables peers to store only important transactions locally. However, the CCF framework does not support advancement proofs. It may be possible to extend incremental Merkle trees to support advancement proofs and to prove a property analogous to EVO-CR. However, the translation is not immediate. An important difference between incremental Merkle trees and deterministic skiplogs is in how hashes are computed. The root hash of the Merkle tree depends directly only on its immediate children, whereas skiplog hashes depend directly on previous values carefully chosen to provide logarithmic-sized advancement proofs.

There have been a number of formal verification efforts related to blockchain consensus [2, 4, 14]. Consensus is orthogonal to proving EVO-CR. The former states a property such as there is a single chain of blocks that is consistent with every honest participant's view, and commits must extend this chain. In contrast, our work entails formal verification of AAOSLs, which can be used to enable verifying claims about past entries in the presence of partial information. In other words, Evolutionary Collision Resistance states that if two peers agree at log index j, even without possessing the entire log, then it is not possible to convince them of conflicting membership claims for indexes $i < j$. The *agree on common* (AOC) property can be related to the universal agreement properties of Pîrlea and Sergey [14] or the common prefix property of Garay et al. [2, 4]. If we consider a set of n "degenerate" advancement proofs, each of which includes only hops between adjacent indexes, starting at genesis and arriving at indexes $k_0, \cdots, k_{n-1}$, respectively, AOC implies that they all agree on all indexes $i \leqslant \min\{k_0, \cdots, k_{n-1}\}$. In our context, AOC is more general because it applies even in the case of partial information: advancement proofs typically do not both visit all indexes.

The most important future work aspect lies in the treatment of hash collisions. We proved our results modulo finding an arbitrary hash collision, similar to Miller et al. [9] or Brun and Traytel [1]. Yet, a full fledged security proof should provide a polynomial time reduction from an authentication failure to a hash collision. We are grateful to Profs. Seny Kamara and José Carlos Bacelar Almeida for useful discussions on this direction.

One promising option to improve our development in this direction is to make our proofs return proof-relevant information about which of its branches evidenced a collision. For example, in the AgreeOnCommon lemma, we could replace the *HashBroke* disjunct as illustrated below.

```
AgreeOnCommon
  : ∀ {j i₁ i₂} {t₁ t₂ : View}
  → {a₁ : AdvPath j i₁} {a₂ : AdvPath j i₂}
  → rebuild a₁ t₁ j ≡ rebuild a₂ t₂ j
  → {i : ℕ} → i ϵ_AP a₁ → i ϵ_AP a₂
  → Either (AOC-Collision t₁ t₂ a₁ a₂)
           (rebuild a₁ t₁ i ≡ rebuild a₂ t₂ i)
```

Here, a value of type *AOC-Collision* t_1 t_2 a_1 a_2 would witness a hash collision from the data provided in its parameters t_1,t_2,a_1 and a_2. The definition of *AOC-Collision* is directly dependent on the structure of AgreeOnCommon, and hence cannot be reused for evo-cr, for instance. The evo-cr would have its own *EvoCR-Collision...* datatype, which would actually use values of *AOC-Collision* because evo-cr calls aoc internally and needs to forward potential collisions. Consequently, every Agda function that returns *Either HashBroke* will have its own collision-tracking datatype, making this a non-trivial effort. Nevertheless, with these *Collision* datatypes at hand, we would explicitly construct the hash collisions and readers could convince themselves of the polynomial runtime constraint based on the structure of these *Collision* datatypes.

Although the approach described above would surely work, it is labor intensive and not transferable. As a future research question, we wonder whether there might be a more automatic way of constructively keeping track of hash collisions. In fact, with a little creativity one could even imagine enforcing the polynomial runtime constraint at the type level using appropriate type systems [5].

6 Concluding Remarks

We have presented formal, machine-checked proofs that Authenticated Append-Only Skip Lists (AAOSLs) indeed provide the properties claimed by its original authors [6].

Our formalization effort also uncovered some improvements to the original AAOSL. For example, providing authenticators separately in *View*s (or with partial maps, as would be done in practice) precludes representing proofs with inconsistent authenticators, eliminating the need to check that, and also results in slightly smaller advancement proofs. Our proofs also highlighted that the actual implementation of *auth* is a red herring. Any definition satisfying *auth-inj-1* and *auth-inj-2* can be used with confidence. This showcases two important aspects of a formal development. On one hand, it uncovers the foundations of the security argument. In this case, we need an injective *auth* function and a definition of hops that do not cross each other. On the other hand, it provides an interactive playground to study variations of the construction. After we finished the development using the definition of *auth* provided by Maniatis and Baker, it was trivial to change it to a simpler variant with confidence that the development is still valid.

This work provides increased confidence that AAOSLs can be used in practice to dramatically reduce the overhead of verifying a recent log state, thus enabling sustainable dynamic participation in authenticated logs such as blockchains. We plan to open-source our development soon.

References

[1] Matthias A. Brun and Dmitriy Traytel. 2019. Generic Authenticated Data Structures, Formally *(Leibniz International Proceedings in Informatics, Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 10. https://doi.org/10.3929/ethz-b-000377525 10th International Conference on Interactive Theorem Proving (ITP 2019); Conference Location: Portland, Oregon, USA; Conference Date: September 8-13, 2019.

[2] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 281–310. https://doi.org/10.1007/978-3-662-46803-6_10

[3] Michael Goodrich and Roberto Tamassia. 2000. *Efficient authenticated dictionaries with skip lists and commutative hashing.* Technical Report. Johns Hopkins Information Security Institute. http://www.cs.brown.edu/cgc/stms/papers/hashskip.pdf

[4] Kiran Gopinathan and Ilya Sergey. 2019. Towards mechanising probabilistic properties of a blockchain. https://popl19.sigplan.org/details/CoqPL-2019/2

[5] Martin Hofmann. 1999. Linear Types and Non Size-Increasing Polynomial Time Computation. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science (LICS '99)*. IEEE Computer Society, USA, 464. https://doi.org/10.1109/LICS.1999.782641

[6] Petros Maniatis and Mary Baker. 2003. Authenticated Append-only Skip Lists. *CoRR* cs.CR/0302010 (2003). http://arxiv.org/abs/cs.CR/0302010

[7] Conor McBride and James McKinna. 2004. The View from the Left. *J. Funct. Program.* 14, 1 (Jan. 2004), 69–111. http://dx.doi.org/10.1017/S0956796803004829

[8] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*, Carl Pomerance (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 369–378. https://dl.acm.org/doi/10.5555/646752.704751

[9] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. 2014. Authenticated Data Structures, Generically. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2535838.2535851

[10] Victor Cacciari Miraldo, Harold Carr, Alex Kogan, Mark Moir, and Maurice Herlihy. 2018. Authenticated Modular Maps in Haskell. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development* (St. Louis, MO, USA) *(TyDe 2018)*. ACM, New York, NY, USA, 1–13. http://doi.acm.org/10.1145/3240719.3241790

[11] Satoshi Nakamoto. 2009. Bitcoin: A peer-to-peer electronic cash system. http://www.bitcoin.org/bitcoin.pdf

[12] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. 2017. CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. 1271–1287. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/nikitin

[13] Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming* (Heijen, The Netherlands) *(AFP'08)*. Springer-Verlag, Berlin, Heidelberg, 230–266. https://doi.org/10.1007/978-3-642-04652-0_5

[14] George Pîrlea and Ilya Sergey. 2018. Mechanising blockchain consensus. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 78–90. https://dl.acm.org/doi/10.1145/3167086

[15] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, et al. 2020. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 983–1002. https://doi.org/10.1109/SP40000.2020.00114

[16] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (June 1990), 668–676. http://doi.acm.org/10.1145/78973.78977

[17] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, et al. 2019. *CCF: A framework for building confidential verifiable replicated services.* Technical Report. Technical Report MSR-TR-2019-16, Microsoft. https://www.microsoft.com/en-us/research/publication/ccf-a-framework-for-building-confidential-verifiable-replicated-services/

[18] Mike J. Spreitzer, Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Douglas B. Terry. 1997. Dealing with Server Corruption in Weakly Consistent, Replicated Data Systems. In *Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking* (Budapest, Hungary) *(MobiCom '97)*. ACM, New York, NY, USA, 234–240. http://doi.acm.org/10.1145/262116.262151