Decentralized Cryptographic Certification and Revocation of Documents

Matt Quinn

CertChain

Abstract. We present a decentralized network consensus protocol for use in cryptographic certification and revocation of documents by institutions with preexisting peer relationships. We refer to the formal specification and model checking results of this protocol to define and demonstrate safety invariance. Details regarding certification status liveness, node security, equivocation scenarios, and applicability to the legal and financial domains follow.

Keywords: document certification and revocation, peering consensus protocol, hashchain replication

1 Introduction

The certification of electronic documents today is done primarily by cryptographically signing an Adobe PDF with a signing certificate issued by a Certificate Authority (CA) participating in Adobe's AATL program. Signing certificates must be purchased from these CAs at substantial cost to the certifier, as the AATL is a centralized PKI administered by Adobe with stringent participation costs and requirements. Revocation of an individual document is not possible; multiple documents are signed with the same certificate, and therefore revocation of a single certificate invalidates the signatures of all documents certified with that certificate.

With the advent of Bitcoin [4], several small companies are offering document certification atop Bitcoin's blockchain: document hashes are assembled into a Merkle tree [3], the root of which is included in an OP_RETURN script and broadcast to nodes on the Bitcoin network for inclusion in the blockchain. Such solutions dissociate the document from its certification status, requiring users to perform an additional verification step that, in practice, few are likely to do or remember. Additionally, these services simply offer point-in-time certifications, for which revocation is not possible.

In the context of higher education, where institutions often certify and occasionally revoke academic records such as diplomas and transcripts, we set out to design and implement a decentralized network on which institutions could certify and revoke documents without the costs and limitations associated with existing commercial solutions. The resulting consensus protocol and implementation of this network, which we name CertChain, allows institutions to do that within the contours of their existing peer relationships.

Each institution participating in a CertChain network maintains a single CertChain network node, configured to listen on the Internet at an advertised TCP/IP hostname and port. Additionally, the node maintains a public/private keypair; from the public key, it derives a Base58 string that it advertises to others as its *address*. Given another institution's triple ⟨hostname, port, address⟩, one can request that institution as a peer. Peers verify, cryptographically sign off on, and keep replicas of, each other's network actions. Each institution groups their actions into blocks, appending them to a non-branching hashchain [5] only after gathering the necessary signatures from their peers. Certification and revocation of documents are two such actions, in addition to the addition and removal of peers.

In the section that follows, we present the formal specification of the consensus protocol underpinning CertChain. This protocol ensures that all of a node's peers agree on the state of a replica, without the need for inter-peer communication and in the face of arbitrary loss, duplication, and reordering of network messages. Our implementation of CertChain is available online[1] as an MIT licensed open source codebase.

2 Consensus Protocol

CertChain's consensus protocol ensures that all of an institution's correct peers maintain a correct replica of the institution's hashchain, without communicating with each other, and in the face of arbitrary loss, reordering, and duplication of network messages. We have written a formal PlusCal [6] specification of this protocol. Throughout this section, we include various definitions from the specification, accompanied by prose descriptions, to describe how CertChain nodes interact with each other. We include key contextual definitions but omit others in the interest of brevity; the specification can be found in its entirety in CertChain's source code repository. In the final subsection, we describe the results of model checking the TLA+ translation of our PlusCal specification.

2.1 Preliminaries

A CertChain node can be described as a PlusCal process in an infinite loop, choosing one of among several atomic steps to execute in each iteration of the loop. All but one of these atomic steps do not involve reception of a network message; when presenting their definitions in this section, we refer to these steps as *daemon steps*. The atomic step involving the reception, and potential processing, of a network message is referred to as the *procmsg step*.

The set of nodes in a CertChain network is defined as the set *Nodes*. Each node maintains several state elements; the following are the key structures that appear in most of the definitions included in this section:

[1] https://gitlab.com/mattjquinn/CertChain

Each node maintains a network input channel.
$channels = [i \in Nodes \mapsto \langle\rangle]$;
Each node maintains their own hashchain.
$hashchains = [i \in Nodes \mapsto \langle\rangle]$;
Each node may maintain replicas of other nodes' chains.
$replicas = [i \in Nodes \mapsto [j \in Nodes \mapsto \langle\rangle]]$;
Each node maintains peering relationships with other nodes.
$peer\_approvals = [i \in Nodes \mapsto [j \in Nodes \mapsto PeeringNotApproved]]$;
Each node gathers signatures for at most one block at a time.
$processing\_block = [i \in Nodes \mapsto None]$;
Each node maintains a queue of actions to include in a new block.
$pending\_actions = [i \in Nodes \mapsto \langle\rangle]$;

The following are valid peer approval states, represented as TLA+ model values:

CONSTANTS *PeeringNotApproved*, *PeeringAwaitingOurApproval*,
PeeringAwaitingTheirApproval, *PeeringApproved*

The following are valid hashchain actions, represented as TLA+ model values:

CONSTANTS *AddPeer*, *RemovePeer*, *Certify*, *Revoke*

The following are valid network messages, represented as TLA+ model values:

CONSTANTS *PeerRequest*,
SignatureRequest, *SignatureResponse*,
BlocksRequest, *BlockManifest*

2.2 Peering

An institution must peer with at least one other institution prior to certifying and revoking documents on the network. To do so, the administrator of an institution's CertChain node issues a PeerRequest to another institution's node; the involved state changes are represented in the following atomic daemon step:

await $Len(pending\_actions[self]) = 0 \land processing\_block[self] = None$;
with ($n \in \{a \in$ DOMAIN $peer\_approvals[self] : a \neq self$
$\land\ peer\_approvals[self][a] = PeeringNotApproved\}$) **{**
$SendNetMsg([mtype \mapsto PeerRequest,\ mfrom \mapsto self],\ channels[n])$;
$peer\_approvals[self][n] := PeeringAwaitingTheirApproval$;
}

If and when the PeerRequest is received and processed by the destination node, it transitions to the following atomic procmsg step conditional:

```
if ( rcvdMsg.mtype = PeerRequest ) {
    if ( peer_approvals[self][rcvdMsg.mfrom] = PeeringNotApproved ) {
        peer_approvals[self][rcvdMsg.mfrom] := PeeringAwaitingOurApproval ;
    } else if ( peer_approvals[self][rcvdMsg.mfrom]
                = PeeringAwaitingTheirApproval ) {
        StmtList 1
        ApprovePeeringRelationship(self, rcvdMsg.mfrom) ;
    } else if ( peer_approvals[self][rcvdMsg.mfrom] = PeeringApproved ) {
        ComputeSignoffPeers(hashchains[self], {}) ;
        if ( rcvdMsg.mfrom ∈ signoff_peers ) {
            StmtList 2
            pending_actions[self] := Append(
                Append(pending_actions[self],
                    {[atype ↦ RemovePeer, node ↦ rcvdMsg.mfrom]}),
                {[atype ↦ AddPeer, node ↦ rcvdMsg.mfrom]}) ;
        } else if ( processing_block[self] ≠ None
            ∧ ∃ a ∈ processing_block[self].actions
                : a.atype = AddPeer ∧ a.node = rcvdMsg.mfrom ) {
            StmtList 3
            SendNetMsg([mtype ↦ SignatureRequest,
                mfrom ↦ self,
                block ↦ processing_block[self]], channels[rcvdMsg.mfrom]) ;
        } else {
            StmtList 4
            ApprovePeeringRelationship(self, rcvdMsg.mfrom) ;
        }
    } else if ( peer_approvals[self][rcvdMsg.mfrom]
                = PeeringAwaitingOurApproval ) {
        skip ;
}
```

If the requested node has not approved a peering relationship with the requester, it simply changes its approval state to PeeringAwaitingOurApproval, after which an administrator will be prompted to approve or deny the request when he or she next logs in to the node's administrative interface. If the node receives a PeerRequest and is already awaiting an approval of its administrator, it simply ignores the request.

Due to arbitrary network behavior, it is possible that the requested node is waiting for the requester to approve the relationship. This can happen if two nodes issue PeerRequests to each other simultaneously. In this case, both nodes will treat the simultaneous issuance as mutual approval, as seen in StmtList 1 above.

It is also possible that the requested node has already approved a peering relationship with the requester. There are three general conditions that predicate the requested node's handling of this situation. First, corresponding to StmtList

2 above, if the requested node already considers the requesting node to be one of its signoff peers, it queues two actions: a RemovePeer action to end the current peering relationship, and an AddPeer action to begin a new one, as requested by the node. Second, corresponding to StmtList 3 above, if the requester is being added as a peer in the requested node's current processing block, the latter sends the former a SignatureRequest for the block. Third, corresponding to StmtList 4, if the requested node does not consider the requester to be its signoff peer, and is not processing a block to add the requester as its peer, it reapproves the peering relationship.

Approval of a peering relationship is defined as follows:

```
macro ApprovePeeringRelationship( self, node_to_approve ) {
  peer_approvals[self][node_to_approve] := PeeringApproved;
  actions := {[atype ↦ AddPeer, node ↦ node_to_approve]};
  pending_actions[self] := Append(pending_actions[self], actions);
}
```

An approving node upgrades its approval of a peering relationship with the requesting node to PeeringApproved, and enqueues an AddPeer action for eventual inclusion in a hashchain block. Requesting and requested nodes share this definition, as illustrated in the daemon step that a node transitions to when an administrator approves of a peering relationship requested by another institution:

```
await Len(pending_actions[self]) = 0 ∧ processing_block[self] = None;
with ( n ∈ {a ∈ DOMAIN peer_approvals[self] : a ≠ self
    ∧ peer_approvals[self][a] = PeeringAwaitingOurApproval} ) {
  ApprovePeeringRelationship(self, n);
}
```

It is important to note that approving a peering relationship does not immediately result in the requester becoming the approver's peer. Only when the approver adds a block to its hashchain containing the AddPeer action does the requester become its peer, and vice versa. Likewise, ending an existing peering relationship is not effective immediately:

```
await Len(pending_actions[self]) = 0 ∧ processing_block[self] = None;
with ( n ∈ {a ∈ DOMAIN peer_approvals[self] : a ≠ self
          ∧ (peer_approvals[self][a] = PeeringApproved
              ∨ peer_approvals[self][a] = PeeringAwaitingOurApproval
              ∨ peer_approvals[self][a] = PeeringAwaitingTheirApproval)} ) {
  If we requested peering, this cancels our request.
  If they requested peering, this denies their request.
  peer_approvals[self][n] := PeeringNotApproved;
  ComputeSignoffPeers(hashchains[self], {});
```

```
  if ( n ∈ signoff_peers ) {
      If we are already peering with them, queue their removal.
    actions := {[atype ↦ RemovePeer, node ↦ n]} ;
    pending_actions[self] := Append(pending_actions[self], actions) ;
   }
 }
```

The addition and removal of peers is effective only upon adding a block containing the appropriate AddPeer or RemovePeer action to one's hashchain. This process is described in the next subsection.

2.3 Adding Blocks

A node can begin the process of adding a new block to its hashchain if it has one or more pending actions and it is not already processing a block. This is represented as the following atomic daemon step:

```
await (processing_block[self] = None ∧ Len(pending_actions[self]) > 0) ;
actions := Head(pending_actions[self]) ;
pending_actions[self] := Tail(pending_actions[self]) ;
ComputeSignoffPeers(hashchains[self], actions) ;
assert self ∉ signoff_peers ;   A node cannot peer with itself.
processing_block[self] :=
    [parent ↦ IF hashchains[self] = ⟨⟩ THEN None
                ELSE hashchains[self][Len(hashchains[self])].uid,
     uid ↦ IF prev_block_uid = None THEN 1 ELSE prev_block_uid + 1,
     author ↦ self,
     actions ↦ actions,
     signatures ↦ [i ∈ signoff_peers ↦ ⟨FALSE, FALSE, FALSE⟩]] ;
prev_block_uid := IF prev_block_uid = None THEN 1 ELSE prev_block_uid + 1 ;
```

After removing the next set of actions from the pending queue, the node computes the set of signoff peers for the block that will include those actions. Informally, the signoff peers for a proposed block that will be appended to a given hashchain consists of those nodes named in AddPeer actions in the chain or the proposed block, provided they are not followed by corresponding RemovePeer actions in the chain or the proposed block. This is formally defined as follows:

```
macro ComputeSignoffPeers( hashchain, actions ) {
  signoff_peers :=
      ({s ∈ Nodes : ∃ b_idx_l ∈ 1 .. Len(hashchain)
          : ∃ a_l ∈ hashchain[b_idx_l].actions
              : a_l.atype = AddPeer ∧ a_l.node = s
          ∧ ∀ b_idx_r ∈ b_idx_l .. Len(hashchain)
              : ¬(∃ a_r ∈ hashchain[b_idx_r].actions
```

$$
\begin{aligned}
&\quad : a\_r.atype = RemovePeer \wedge a\_r.node = s)\} \\
&\cup \{s \in Nodes : \exists\, m \in actions : m.atype = AddPeer \quad \wedge m.node = s\}) \\
&\setminus \{s \in Nodes : \exists\, m \in actions : m.atype = RemovePeer \wedge m.node = s\}
\end{aligned}
$$

}

The node then creates a new processing block. Each of the signoff peers is required to cryptographically sign this block before the node can append it to its hashchain; For specification purposes, whether or not a peer's signature is present on the block is represented by the second Boolean element of its corresponding 3-tuple in the *signatures* field of the block. In our implementation of CertChain, this field consists of actual ECDSA signatures issued by each node. Likewise, our specification assigns each block a monotonically increasing *uid* from $\mathbb{N}$, whereas our implementation considers the probabilistically unique output of the SHA256 hash function on a portion of the block to be the block's unique identifier.

After creating a block, the node sends a SignatureRequest to each of the block's signoff peers in an atomic daemon step per peer:

$$
\begin{aligned}
&\textbf{await } (processing\_block[self] \neq None)\,\textbf{;} \\
&\textbf{with } (\ peer \in \{s \in \text{DOMAIN } processing\_block[self].signatures \\
&\quad : \neg processing\_block[self].signatures[s][1]\}\ )\ \textbf{\{} \\
&\ \ SendNetMsg([mtype \mapsto SignatureRequest, \\
&\qquad\qquad mfrom \mapsto self, \\
&\qquad\qquad block \mapsto processing\_block[self]],\ channels[peer])\,\textbf{;} \\
&\ \ processing\_block[self].signatures[peer][1] := \text{TRUE}\,\textbf{;} \\
&\textbf{\}}
\end{aligned}
$$

If and when a peer receives a SignatureRequest, it transitions to the following atomic procmsg step conditional:

$$
\begin{aligned}
&\textbf{if } (\ rcvdMsg.mtype = SignatureRequest \\
&\quad \wedge (peer\_approvals[self][rcvdMsg.mfrom] = PeeringApproved \\
&\qquad \vee (peer\_approvals[self][rcvdMsg.mfrom] \\
&\qquad\quad = PeeringAwaitingTheirApproval \\
&\qquad\ \wedge \exists\, action \in rcvdMsg.block.actions \\
&\qquad\quad : action.atype = AddPeer \wedge action.node = self)) \\
&\quad \wedge \neg(\exists\, b\_idx \in \text{DOMAIN } replicas[self][rcvdMsg.block.author] \\
&\qquad : replicas[self][rcvdMsg.block.author][b\_idx].parent \\
&\qquad\quad = rcvdMsg.block.parent) \\
&\quad \wedge \neg(\exists\, sr\_idx \in \text{DOMAIN } sigreqs\_pending\_sync[self] \\
&\qquad : sigreqs\_pending\_sync[self][sr\_idx].block.uid \\
&\qquad\quad = rcvdMsg.block.uid \\
&\qquad \wedge sigreqs\_pending\_sync[self][sr\_idx].block.author \\
&\qquad\quad = rcvdMsg.block.author)\ )\ \textbf{\{} \\
&\ \ \textbf{if } (\ peer\_approvals[self][rcvdMsg.mfrom] \\
&\qquad = PeeringAwaitingTheirApproval\ )\ \textbf{\{}
\end{aligned}
$$

```
    ApprovePeeringRelationship(self, rcvdMsg.mfrom);
  };
  RequestBlocksIfParentIsAbsent(rcvdMsg,
      replicas[self][rcvdMsg.mfrom], were_blocks_requested);
  sync_or_sign: if ( were_blocks_requested ) {
    sigreqs_pending_sync[self]
        := Append(sigreqs_pending_sync[self], rcvdMsg);
  } else {
    IssueSignatureResponse(self, rcvdMsg);
  }
}
```

Three conjunctive predicates must be satisfied in order for a node to process a SignatureRequest. The first requires that the receiver has already approved a peering relationship with the sender, or is awaiting the sender's approval, in which case the block to be signed must contain an AddPeer action naming the receiver and therefore approving a peering relationship with the receiver. The second requires that the block does not claim as its parent a block in the sender's hashchain that has already been claimed, as CertChain hashchains are not allowed to branch. The third requires that the SignatureRequest is not already queued for processing by the receiver pending block synchronization (detailed below).

If the first disjunct of the first conjunct applies, the first statement in the list marks the peering relationship with the sender as approved now that the sender has approved the receiver's request. Following that, and irrespective of the applicable disjunct, the receiver determines if the tail node of its replica of the sender's hashchain is the parent block of the block to be signed:

```
macro RequestBlocksIfParentIsAbsent( rcvdMsg,
    replica, were_blocks_requested ) {
  if ( (replica = ⟨⟩ ∧ rcvdMsg.block.parent ≠ None)
      ∨ (replica ≠ ⟨⟩ ∧ replica[Len(replica)].uid ≠ rcvdMsg.block.parent) ) {
    SendNetMsg([mtype ↦ BlocksRequest,
                mfrom ↦ self,
                after_block_uid ↦ IF Len(replica) = 0
                    THEN None ELSE replica[Len(replica)].uid,
                next_block_uid ↦ IF Len(replica) = 0
                    THEN None ELSE replica[Len(replica)].uid],
               channels[rcvdMsg.block.author]);
    were_blocks_requested := TRUE;
  } else {
    were_blocks_requested := FALSE;
  }
}
```

If the receiver does not have the block's parent, it needs to synchronize its replica of the sender's hashchain with the sender before it can issue a signature for the block. The sender asks the receiver to send all of the blocks in its chain after the last block in its replica, using the model value *None* if its replica is empty, in the form of a BlocksRequest. It then queues the SignatureRequest for later processing, which will occur if and when it receives all of the blocks up to and including the parent of the block to be signed. Otherwise, if the receiver does have the block's parent, it signs the block and sends its signature to the sender in the form of a SignatureResponse:

```
macro IssueSignatureResponse( self, sigreq ) {
  SendNetMsg([mtype ↦ SignatureResponse,
              mfrom ↦ self,
              signed_block_uid ↦ sigreq.block.uid],
             channels[sigreq.mfrom]);
}
```

As mentioned earlier, our implementation requires that SignatureResponses contain an actual ECDSA signature that can be verified; this definition uses the simple *signed_block_uid* field instead for modeling purposes.

As a node receives SignatureResponses from its peers, it accumulates them on the processing block, as defined in the following atomic procmsg step conditional:

```
if ( rcvdMsg.mtype = SignatureResponse ) {
  if ( processing_block[self] ≠ None
       ∧ rcvdMsg.signed_block_uid = processing_block[self].uid ) {
    processing_block[self].signatures[rcvdMsg.mfrom][2] := TRUE;
  }
}
```

Once a node has acquired all of the necessary signatures for a block, it sends the block, with all acquired signatures included, to each of the peers who signed. Each transmission is represented as an atomic daemon step:

```
await (processing_block[self] ≠ None
    ∧ ∀ s ∈ DOMAIN processing_block[self].signatures
        : processing_block[self].signatures[s][1]
        ∧ processing_block[self].signatures[s][2]);
with ( peer ∈ {s ∈ DOMAIN processing_block[self].signatures
    : ¬processing_block[self].signatures[s][3]} ) {
  SendNetMsg([mtype ↦ BlockManifest,
              block ↦ processing_block[self]], channels[peer]);
  processing_block[self].signatures[peer][3] := TRUE;
} ;
```

The receipt and processing of BlockManifests is detailed in the next subsection. Once a node has sent the signed block to all of its peers, it appends the block to its hashchain in the following atomic daemon step:

await ($processing\_block[self] \neq None$
$\quad \wedge \forall\, s \in \text{DOMAIN}\ processing\_block[self].signatures$
$\quad\quad : processing\_block[self].signatures[s][1]$
$\quad\quad \wedge\ processing\_block[self].signatures[s][2]$
$\quad\quad \wedge\ processing\_block[self].signatures[s][3])$ **;**
$AppendBlockToChain(hashchains[self],$
$\quad processing\_block[self],\ hashchains\_block\_uid\_set[self])$ **;**
$processing\_block[self] := None$ **;**

where appending a block to a hashchain is defined as follows:

macro *AppendBlockToChain*(*chain*, *block*, *block_uid_set*) **{**
assert $(chain = \langle\rangle) \equiv (block.parent = None)$ **;**
assert $chain = \langle\rangle \vee chain[Len(chain)].uid = block.parent$ **;**
$chain := Append(chain,\ block)$ **;**
$block\_uid\_set := block\_uid\_set \cup \{block.uid\}$ **;**
}

This definition also applies to the appending of blocks to hashchain replicas by a node's peers, as detailed in the next subsection.

2.4 Replica Synchronization

As shown in the previous subsection, a node does not immediately respond to a SignatureRequest if it does not have the parent block of the block to be signed. As nodes initiate new peering relationships with each other, they will need to synchronize their replicas of each other's hashchains prior to signing a block claiming a parent that was authored prior to the start of the relationship. When this occurs, a node queues the SignatureRequest and issues a BlocksRequest to the author of the hashchain replica in question. Upon receiving a BlocksRequest, a receiving node first queues the request, as represented by the following atomic procmsg step conditional:

if ($rcvdMsg.mtype = BlocksRequest$ **) {**
$blocks\_request\_queues[self] := Append(blocks\_request\_queues[self],\ rcvdMsg)$ **;**
}

Recall that the sender of a BlocksRequest provides the unique identifier of the block at the tip of their replica. The receiver iterates through the sequence of blocks in their hashchain following that block, sending each block to the sender in a BlockManifest. A single such iteration is represented by the following atomic

daemon step:

```
await Len(blocks_request_queues[self]) > 0 ;
block_to_send := hashchains[self][CHOOSE b_idx ∈ 1 .. Len(hashchains[self])
    : hashchains[self][b_idx].parent
        = blocks_request_queues[self][1].next_block_uid] ;
SendNetMsg([mtype ↦ BlockManifest,
            block ↦ block_to_send],
           channels[blocks_request_queues[self][1].mfrom]) ;
if ( block_to_send = hashchains[self][Len(hashchains[self])] ) {
  blocks_request_queues[self] := Tail(blocks_request_queues[self]) ;
 } else {
   blocks_request_queues[self][1].next_block_uid := block_to_send.uid ;
 } ;
```

As seen in this definition, iteration starts at the block immediately succeeding the unique identifier of the block specified in the BlocksRequest. Ideally, the resulting sequence of transmissions to the requester would therefore satisfy the invariants asserted in the AppendBlockToChain definition. In practice, we must account for the possible loss and reordering of network messages, as seen in our definition for the processing of BlockManifests as an atomic procmsg step conditional:

```
if ( rcvdMsg.mtype = BlockManifest
    ∧ ¬(∃ b_idx ∈ DOMAIN replicas[self][rcvdMsg.block.author]
        : replicas[self][rcvdMsg.block.author][b_idx].uid = rcvdMsg.block.uid)
    ∧ ¬(∃ mf_idx ∈ DOMAIN manifests_pending_sync[self]
        : manifests_pending_sync[self][mf_idx].block.uid = rcvdMsg.block.uid
        ∧ manifests_pending_sync[self][mf_idx].block.author
            = rcvdMsg.block.author) ) {
  assert ∀ s ∈ DOMAIN rcvdMsg.block.signatures
      : rcvdMsg.block.signatures[s][2] ;
  RequestBlocksIfParentIsAbsent(rcvdMsg,
      replicas[self][rcvdMsg.block.author], were_blocks_requested) ;
  if ( were_blocks_requested ) {
    manifests_pending_sync[self]
        := Append(manifests_pending_sync[self], rcvdMsg) ;
   } else {
     AppendBlockToChain(replicas[self][rcvdMsg.block.author],
         rcvdMsg.block, replicas_block_uid_set[self][rcvdMsg.block.author]) ;
   }
 }
```

Assuming the satisfaction of the two conjunctive predicates requiring that the block within the received BlockManifest not already have been appended to the

replica or be queued for later processing, the block is appended if its parent block is the tip of the node's replica. Otherwise, just as SignatureRequests are queued if the parent of the block to be signed is absent, BlockManifests that cannot immediately be appended are held in their own queue and a BlocksRequest is sent to the hashchain's author.

As replica synchronization progresses at a node, it continually checks both its pending SignatureRequest and BlockManifest queues and responds or appends to its replica, respectively, those claiming a parent block that is now present in the replica. We represent both checks as atomic daemon steps. These definitions are straightforward and similar, so we omit both here; they can be found in the full specification as mentioned above.

2.5 Certifying and Revoking Documents

As long as an institution has at least one peer, it may certify and revoke documents. Certification is represented as an atomic daemon step in our specification as follows:

await $Len(pending\_actions[self]) = 0 \land processing\_block[self] = None$;
$ComputeSignoffPeers(hashchains[self], \{\})$;
await $signoff\_peers \neq \{\}$;
$actions := \{[atype \mapsto Certify,\ doc\_id \mapsto next\_doc\_id]\}$;
$next\_doc\_id := next\_doc\_id + 1$;
$pending\_actions[self] := Append(pending\_actions[self],\ actions)$;

This definition only certifies a single document per step for simplicity; in practice, and in our implementation, multiple documents can be certified in a single set of actions, and thus within a single hashchain block. Also for specification simplicity, we represent the identifier of a document, *doc_id*, using a monotonically increasing sequence from $\mathbb{N}$; in our implemenation, the text of the document being certified is combined with a random nonce and provided as input to the SHA256 hash function, with the resulting output used as the *doc_id*.

A document is not considered certified until the certifying institution appends a block to its hashchain containing a certification action naming the document by its identifier. This requires the signatures of an institution's peers, as detailed in preceding subsections. Once a document has been certified, an institution can revoke it, as represented in the following atomic daemon step:

await $Len(pending\_actions[self]) = 0 \land processing\_block[self] = None$;
$ComputeSignoffPeers(hashchains[self], \{\})$;
await $signoff\_peers \neq \{\}$;
$ComputeCertifiedDocIds(hashchains[self], (next\_doc\_id - 1))$;
await $certified\_doc\_ids \neq \{\}$;
$actions := \{[atype \mapsto Revoke,\ doc\_id \mapsto \text{CHOOSE } id \in certified\_doc\_ids : \text{TRUE}]\}$;
$pending\_actions[self] := Append(pending\_actions[self],\ actions)$;

As with certifications, our specification models a single revocation per step for simplicity; our implementation allows multiple revocations per block. The set of certified, and thus revocable, documents is defined as follows:

$$
\begin{array}{l}
\textbf{macro } \mathit{ComputeCertifiedDocIds}(\ \mathit{hashchain}, \mathit{last\_doc\_id}\ )\ \{ \\
\mathit{certified\_doc\_ids} := \\
\quad \{\mathit{id} \in 1\,..\,\mathit{last\_doc\_id} : \exists\, \mathit{b\_idx} \in \text{DOMAIN } \mathit{hashchain} \\
\qquad : \exists\, a \in \mathit{hashchain}[\mathit{b\_idx}].\mathit{actions} : a.\mathit{atype} = \mathit{Certify} \wedge a.\mathit{doc\_id} = \mathit{id}\} \\
\quad \setminus \{\mathit{id} \in 1\,..\,\mathit{last\_doc\_id} : \exists\, \mathit{b\_idx} \in \text{DOMAIN } \mathit{hashchain} \\
\qquad : \exists\, a \in \mathit{hashchain}[\mathit{b\_idx}].\mathit{actions} : a.\mathit{atype} = \mathit{Revoke} \wedge a.\mathit{doc\_id} = \mathit{id}\} \\
\}
\end{array}
$$

Notably, upon reception of a SignatureRequest for a block, peers can apply this definition to their replica of the authoring institution's hashchain to ensure that the authoring institution is not re-certifying a document that has already been revoked. This is just one of several equivocation scenarios that peers can detect as a result of the invariants established by our consensus protocol; additional scenarios are detailed below.

2.6 TLA+ Model Checking Results

We performed two model checking executions on the TLA+ translation of our PlusCal specification using TLC, the TLA+ model checker. In addition to the invariant assertions found in several of the definitions presented above, we define the overarching safety invariant for our consensus protocol as follows:

$$
\begin{array}{l}
\mathit{SafetyInvariant} \triangleq\ \forall\, n \in \{a \in \mathit{Nodes} : \mathit{hashchains}[a] \neq \langle\rangle\} \\
\quad : \forall\, s \in \text{DOMAIN } \mathit{hashchains}[n][\mathit{Len}(\mathit{hashchains}[n])].\mathit{signatures} \\
\qquad : \mathit{replicas\_block\_uid\_set}[s][n] = \mathit{hashchains\_block\_uid\_set}[n] \\
\qquad \vee (\mathit{processing\_block}[n] \neq \mathit{None} \wedge \mathit{replicas\_block\_uid\_set}[s][n] \\
\qquad\quad \setminus \mathit{hashchains\_block\_uid\_set}[n] = \{\mathit{processing\_block}[n].\mathit{uid}\}) \\
\qquad \vee \mathit{hashchains\_block\_uid\_set}[n] \setminus \mathit{replicas\_block\_uid\_set}[s][n] \\
\qquad\quad = \{\mathit{hashchains}[n][\mathit{Len}(\mathit{hashchains}[n])].\mathit{uid}\}
\end{array}
$$

This invariant defines safety such that at any reachable configuration of node states in a CertChain network, the hashchain replica maintained by each peer of any given node is: identical to the node's hashchain; or, ahead only by the block being processed by the node; or, missing only the latest block in the node's hashchain. We define safety in this manner as a formalization of the requirement that all of a node's peers maintain replicas differing from the node's hashchain, and therefore each other's replicas, by at most the latest block authored by the node, without inter-peer coordination and despite network inconsistencies and failures.

We provided this invariant to TLC for enforcement, and additionally specified that TLC check for reachable configurations leading to deadlock, where one or more nodes is unable to transition to another step. Two breadth-first search

executions of TLC were performed, the first with a set of three nodes and the second with a set of two, both with a state constraint limiting the length of each node's hashchain to 100 and each node's network buffer to three. Due to state space explosion and subsequent resource limitations on the TLC workstation, both executions had to be aborted before the entire state space graph could be constructed. The following table shows the model checking status of both executions prior to termination:

Nodes	States Found	Distinct States	Diameter	Queued States	100% Spec Coverage?
3	14.8 billion	2.9 billion	22	1.6 billion	No
2	21 billion	5.8 billion	35	2.1 billion	Yes

Both executions were performed on separate AWS EC2 i2.4xlarge instances possessing 16 2.5GHz virtual CPUs, 122 GiB of RAM, and a local 800 GB SSD; the three-node and two-node executions spanned 18 and 23 hours of real time, respectively. Although neither execution completed an exhaustive state space search, for each of the several billion state configurations it did reach, TLC found no violations of the definition, safety, and deadlock invariants we provided. Additionally, during the two-node execution, every step in our specification was reached in at least one state configuration, yielding 100% specification coverage.

3 Third-Party Determination of Document Status

When a third party, which we define as any individual or institution other than an authoring CertChain institution and its peers, wishes to verify the certification status of a document authored by the institution, the simplest and most common way to do so is to provide the identifier of the document to the authoring institution and request proof of certification status and liveness. We consider this the simplest because the third party does not have to possess the contents of the document at the time of request, as the authoring institution maintains a repository of the documents it has certified and can return the contents of the document along with the status and liveness proof. The status proof is provided in the form of a Merkle inclusion proof: the hash of a tuple containing the document identifier and its status are included in a Merkle tree, with the proof demonstrating inclusion of this tuple in a Merkle tree containing corresponding tuples for every document authored by the institution. The block header of the authoring institution's most recent block serves as the liveness proof: if the Merkle inclusion proof yields the Merkle tree root found in this most recent block header, where recency is determined in relation to a fixed-length time epoch in a manner similar to that of CONIKS [2], the third party can confirm the document's status.

In practice, few third parties have an interest or reason to be familiar with the status and liveness proofs associated with the document they are verifying. Our implementation of CertChain addresses this, without offloading the task of

verification to another entity, by performing proof assertions and integrity checks in the third party's browser using the Stanford JavaScript Crypto Library [8] and a JavaScript transcompilation of the secp256k1 C++ library. As of this writing, our implementation uses these libraries to assert that all of the following are true when verifying the status of a document:

- Does the document hash to the expected and received hash values?
- Is the header of the latest block signed by the authoring institution?
- Are there one or more peers identified as signatories in the block header?
- Is the block header signed by all peers identified in the header?
- Are author and peers in agreement about each other's address, hostname, and port?
- Does the provided Merkle inclusion proof tie this document to the signed root?
- Was this block published within the last hour?

With an eye towards use in certifying and revoking academic records, we designed CertChain to allow third parties to verify the status of a document in perpetuity, even if the certifying institution is defunct. We have shown how our consensus protocol ensures peer-wide replica consensus without inter-peer coordination. It then follows that a third party may choose one of an institution's peers, rather than the institution itself, to verify the status of a document certified by the institution. Although our implementation does not yet support this mechanism, adding it is simply a matter of allowing users to download raw documents from the authoring institution and implementing an additional web application endpoint such that users can choose one of the insititution's peers and supply either the raw document or a link to the raw document to be verified. Notably, neither choice requires the document itself to be transmitted to the chosen peer; only the document's identifier need be transmitted. The same client-side proof assertions and integrity checks detailed above would then be applied to the peer's response.

4 Node Security and Equivocation

If an attacker gains read-only access to a CertChain node's filesystem, the node's keypair and all of the documents certified by the node are considered compromised. Although the attacker can install the keypair in his own CertChain node and ask the node's peers to sign forged blocks, the SignatureResponses will be sent to the node, not the attacker. When the node receives SignatureResponses for a block it is not currently processing, it simply ignores them, as detailed above. Thus, with read-only access to a node's keypair, it is not possible for an attacker to certify forged documents such that they are included in the node's hashchain or in the replicas maintained by its peers.

Should an attacker additionally gain write access to the CertChain node's filesystem, she may modify the contents of a previously certified document. Such a modification will be detected during client-side verification, as detailed above,

when the hash of the modified document does not match that of the original document for which the Merkle inclusion proof applies. The attacker may attempt to prevent this by modifying the Merkle tree root of the hashchain block in which the original document was certified; she must also re-sign the block with the node's keypair, and generate her own keypairs for use in forging the signatures of the node's peers. This attack is only effective on the most recent block, as it is the block used to prove status liveness. If, and as long as, the block is at the tip of the node's chain, and is thus the most recent block, it will be used to prove liveness to third parties, as detailed above, and the attacker can distribute the modified document's identifier to others knowing it will be presented to them as certified. However, if the node uses this modified block as the parent of its next block, the SignatureRequests it sends to its peers will be queued pending replica synchronization. Upon being sent the modified parent block, peers will reject it: a peer that has the correct block will see that the modified block claims an already-claimed parent, while peers that don't will proceed to inspect the modified block and see that their signatures have been forged. In both cases, peers will refuse to add the modified block to their replicas, and thus will never respond to any queued SignatureRequest for a block that claims it as a parent.

Administrative access to the machine on which a CertChain node is running allows an attacker to modify certified documents, forge new certifications, and forge new revocations without peer rejection, provided that she is able to replace the node's peers with ones she controls so as to prevent them from receiving modified blocks. She may also introduce arbitrary modifications to the network node's behavior. However, she will not be able to withhold signatures from peers such that they are unable to make progress in adding new blocks to their chains; those peers can simply end their peering relationship with the compromised node, an action which does not require the latter's signature. Critically, without administrative access to a node's peers, an attacker cannot modify or destroy correct history replicas; as long as a node's hashchain can be compared to a single correct replica, evaluation of the consensus protocol safety invariant we specified above enables any third party to determine if a node is equivocating.

5 Application Domains

Our approach to inter-institutional document certification and revocation has applications beyond academic records: any domain with extensive document certification needs, including but especially the legal profession, should consider the consensus protocol and implementation we have described here. CertChain enables any group of related entities to harness their existing peer relationships for use in certifying and revoking documents, without the costs imposed by centralized certification service vendors.

As of this writing, the financial establishment's marked interest in decentralized transaction settlement has been narrowly inspired by Bitcoin's particular implemenation [9]. Notable alternatives dissociate themselves from Bitcoin

proper while retaining global consensus as an operational requirement, i.e., Ripple's collectively-trusted subnetworks [7] and Stellar's quorum intersection [1]. For purposes of transaction settlement among peer financial institutions, neither proof-of-work nor global consensus is needed to settle transactions involving a subset of entities whose identities are known. Our description of CertChain and its consensus protocol are more naturally applicable to this and other transactional use cases involving inherently stratified network participants.

References

1. Mazières, David. "The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus."
2. Melara, M. S., Blankstein, A., Bonneau, J., Freedman, M. J., & Felten, E. W. (2014). *CONIKS: A privacy-preserving consistent key service for secure end-to-end communication.* Cryptology ePrint Archive, Report 2014/1004.
3. Merkle, Ralph C. "A digital signature based on a conventional encryption function." *Advances in Cryptology-CRYPTO'87*. Springer Berlin Heidelberg, 1988.
4. Nakamoto, Satoshi.: "Bitcoin: A peer-to-peer electronic cash system." *Consulted* 1.2012 (2008): 28.
5. Lamport, Leslie. "Password authentication with insecure communication." *Communications of the ACM* 24.11 (1981): 770-772.
6. Lamport, Leslie. "The PlusCal algorithm language." *Theoretical Aspects of Computing-ICTAC 2009*. Springer Berlin Heidelberg, 2009. 36-60.
7. Schwartz, David, Noah Youngs, and Arthur Britto. "The Ripple protocol consensus algorithm." *Ripple Labs Inc White Paper* (2014).
8. Stark, Emily, Michael Hamburg, and Dan Boneh. "Symmetric cryptography in javascript." *Computer Security Applications Conference, 2009. ACSAC'09. Annual.* IEEE, 2009.
9. Swanson, Tim. "Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems."