CellTree: A New Paradigm for Distributed Data Repositories

Anasuya Acharya* IIT Bombay
Manoj Prabhakaran* IIT Bombay
Akash Trehan* IIT Bombay

May 17, 2019

Abstract

We present CellTree, a new architecture for distributed data repositories. The repository allows data to be stored in largely independent, and highly programmable *cells*, which are *"assimilated"* into a tree structure. The data in the cells are allowed to change over time, subject to each cell's own policies; a cell's policies also govern how the policies themselves can evolve. A design goal of the architecture is to let a CellTree evolve organically over time, and adapt itself to multiple applications. Different parts of the tree may be maintained by different sets of parties interested in the respective parts, and the core mechanisms used for maintaining the tree can also vary across the tree and over time.

We present provable guarantees of liveness, correctness and consistency (the last one being a generalization of the typical blockchain guarantee of "persistence," when data is dynamic), when the CellTree architecture is instantiated using a simple set of modules. These properties can be guaranteed for individual cells that satisfy requisite trust assumptions, even if these trust assumptions do not hold for other cells in the tree.

We also discuss several features of a CellTree that can be exploited by applications. We leave it for future work to develop full-fledged applications on top of this powerful architecture.

1 Introduction

There has been an explosion of interest in the notion of a *distributed ledger*, triggered by the popularity of Bitcoin [14]. The typical distributed ledgers today have the form of a *blockchain*, where each new block points to an earlier block. A variety of ingenious protocols have been developed to add and *immutably* maintain the blocks in such a ledger in an adversarial environment, while incentivizing participation. Blockchain applications today have gone well beyond that envisaged by Bitcoin (namely, to publicly store all the transactions of a cryptocurrency), and have been used (or proposed for use) as ledgers for supply chains, in gaming, for maintaining public records, and for general purpose contracts and transactions.

However, as blockchains have exploded in popularity and scope, several issues have come to the fore and – despite several proposed alternatives – many of them remain unresolved. Typical blockchain architectures face scalability challenges, as the underlying blockchain structure is ever-growing and all the "full nodes" need to store this entire chain to be able to fully validate *any* block. Further, this data may include illegal content, creating legal complications for the blockchains [11, 15]. Implementation bugs, if exploited, can create irreversible effects, thanks to the immutable nature of a blockchain. Another issue of note is that the proof-of-work consensus protocol used in popular blockchains have turned out to be ecologically costly.

Many recent works have proposed fixes to these – using secondary structures [1, 3] to reduce the content stored on a blockchain, using alternate consensus mechanisms [2, 7, 9], and using different graph topologies [4, 16]. However, as these systems grow in scale, or if new attacks emerge due to changing economic incentives or discovery of new bugs, the deployments of these solutions may also need revisions.

In this work, we propose a very different approach to the problem of designing a large scale, long-running system for distributed data storage. A distributed data repository is a complex system, with several constituent components, addressing several disparate sub-problems. A key philosophy of our approach is to *let different solutions coexist* in the system, and to *leave room for the system to organically evolve* over time and across applications. For this, various sub-problems are delegated to *modules*, and the overall architecture is agnostic to how each module is implemented.

In identifying the modules, we delineate tasks that are typically conflated together in current approaches. For instance, in a typical blockchain, selecting the contents for a new cell, and confirming the cell as part of a blockchain are usually accomplished by a single mechanism that serves both as a lottery system and a consensus protocol; in our design, these two tasks are assigned

*Author names sorted alphabetically.

to separate modules, and typically separate sets of parties are responsible for the two.

An important consideration in our design is to let different users of the system focus on different parts of the system, unencumbered by the entire system's data. A related feature is the ability of a part of the whole structure to function as a smaller version of the system, complete with its security guarantees and trust assumptions. In particular, we naturally admit *multi-level confirmation* of new blocks, so that clients trusting lower levels in this hierarchy can get quick confirmation of the addition of a block to the structure, and those who do not trust those levels can wait for a higher level of confirmation.

The CellTree Architecture. We term our new design *CellTree*, contrasting it with a blockchain. A CellTree consists of largely independent *cells*, with their own rules for collecting and updating data. As mentioned above, the data in each cell can evolve subject to policies programmed into the cell, and even these policies can evolve as they permit themselves to. Each cell is "operated" by a crew selected for it (using a selection procedure that is encapsulated into a module that may vary across cells).

The cells act collectively to provide various guarantees. Each cell is addressed by associating it with a node in a tree. Data in the different cells are periodically *assimilated* into the tree. Each cell's integrity and availability guarantees would depend only on (the crews of) the cells in the path from it to the root of the tree (or to any node whose crew is considered to have an honest-majority). Subtrees could be easily excised from the CellTree, or grafted to multiple locations (effectively making the structure a directed acyclic graph, rather than a tree), with little effect on cells outside the subtree.

The cell structure and the tree structure are complementary, and it is primarily the interface between them that is fixed by the CellTree architecture. The individual cells themselves can be programmed to evolve in customized ways, and the tree's protocols can also be customized using various modules at the level of nodes, edges and paths.

Organization. The rest of this paper is organized as follows. We start with a high-level discussion of the design goals and some features in Section 2. Section 3 introduces the different cryptographic primitives and notation used in the design. Section 4 and Section 5 present the technical details of a cell and the tree, respectively, and Section 5.2 lists the various external modules that need to be plugged into the design to instantiate a CellTree. Section 6 outlines a simple instantiation of the CellTree architecture, and presents formal guarantees that can be given for this instantiation. Several features (not exploited in our example) are discussed in Section 6.3. Before concluding, we discuss a few related constructions.

2 An Organic Design

Before we present the technical details of our construction, we pause to reflect on our key design goals: We seek a flexible distributed repository, where multiple solutions to various sub-problems can co-exist, and even evolve over time, to address different use-cases and attack scenarios. We require the framework to be rich enough to directly support a variety of conventional applications of distributed ledgers (and more), but should not be over-engineered so that it becomes too rigid for unforeseen future applications. The CellTree framework strives to meet this goal by being *modular*, *cellular* and *evolving*, as sketched below.

Modular We separate out processes (a) for assigning responsibilities to parties, (b) for selecting updates to be applied to the data in the system, and (c) for parties to enforce integrity and availability of the data they are responsible for. Our focus in this work will be on (c), with support for a wide variety of options for the modules for (a) and (b). Indeed, concepts from prior blockchain constructions, like proof-of-work, proof-of-stake and the use of crypto currencies as incentives, as well as relying on permissioned systems, all remain possible means to achieving the goals of (a) and (b) in our system. Even in our solution for (c), various sub-tasks are left as modules, which can be implemented in a variety of ways to achieve different efficiency-robustness trade-offs.

Cellular We generalize the notion of blocks in a blockchain to add more functionality and flexibility. For clarity, we use the term *cell* instead of block. The system is designed so as to allow cells to operate somewhat independent of each other, while retaining a cohesive structure overall. This independence manifests in different ways:

- As time progresses more cells can be added to the system. Cells could also be excised from the system, with limited impact on the rest of the system.
- The parties hosting the distributed system can choose to focus on cells of interest to them, and ignore the remaining.
- Each cell may use its own set of mechanisms for implementing the modules mentioned above.

Evolving We envisage cells as having static addresses, but their data content could change over time. Data updates in a cell must follow an *evolution policy* associated with the cell. Simple examples of evolution policy include one that enforces that the data remains static and one that only allows blocks to be appended to a list of data blocks. The evolution policy in a cell itself can evolve, and this is also dictated by the evolution policy.[1]

There is one more sense in which a CellTree can evolve: the modules used by various nodes in the tree can evolve, subject to their own evolutionary policies. However, we do not mandate a framework for how modules are chosen and modified, instead leaving this itself to a module.

2.1 Design Overview

The basic building block of a CellTree is a *cell*, which carries the data, as well as a (typically smaller) *nucleus*. As detailed in Section 4, the nucleus has code that specifies how a cell can evolve (e.g., cell data can only be appended to).

The cell is hosted in a *node* in a binary tree. Each node is operated by a *crew*. Each node's crew is also in charge of *monitoring* the nodes in a relatively small subtree rooted at that node. Monitoring involves verifying that the evolution of a nucleus is consistent with its own policies. If so, the monitoring node *assimilates* the updated nucleus into its own hash pointer, which is then propagated to its own ancestors. Further, it also propagates a *proof of assimilation* towards the nodes it is monitoring. These proofs will be verified by a client accessing a cell.

We specify several algorithms executed by the crew members operating a node (or clients accessing a node), to carry out a cell's evolution, to monitor the evolution of some other cells, and to assimilate updates and propagate assimilation signals up and down the tree. These algorithms are given in Section 5. Many tasks like selecting the next step in a cell's evolution, selecting crews for new nodes, communicating to other crews, etc. are left to the *modules* listed in Section 5.2.

2.2 CellTree vs. Blockchains

A conventional blockchain, like that of Bitcoin, could be viewed as a single node CellTree, with an appropriately programmed cell, and a very large crew. Conversely, a CellTree has many parallels with a blockchain, with cells in lieu of blocks. But the CellTree architecture differs from blockchains in several important and fundamental ways. Below, we discuss a few such differences.

Multi-Level Confirmation. In a CellTree, a crew operating a node can update its cell autonomously, based on its own local policies, and then have it assimilated into the tree. Unlike in a blockchain architecture, where the whole system has to approve the update, the updates within a cell are wholly determined by the crew of the node hosting the cell (subject to the policies programmed into the cell); a client that trusts the node's crew has *immediate confirmation* of the update. The purpose of assimilation and higher levels of confirmation is only to protect against (and hence disincentivize) misbehaving crews.

Reversed Hash Pointers. One of the easily spotted difference between the CellTree architecture and blockchains is that the former uses a tree topology, instead of a chain.

At first glance, one may view a blockchain as a *unary* Merkle Tree [12, 13] (i.e., each node has a single child), while a CellTree uses a binary Merkle tree. However, this comparison is misleading. In a CellTree, the "hash pointers" point from a parent (older node) to its children (newer nodes), whereas in a blockchain, newly added nodes carry hash pointers to existing nodes.[2] That is, if we view a blockchain as a unary Merkle tree, the "genesis" block would play the role of a leaf, and the most recently added block becomes the root. The reversed direction of hash pointers reflects the fact that in a conventional blockchain, a block gets confirmed when future blocks attach to it, whereas in a CellTree, a cell (or an update to a cell) is confirmed by already existing nodes.

Distributed Ownership. An alternate attempt to relate the CellTree structure to a blockchain would be to compare it to a blockchain that "forks" often (and retains the forks). This analogy too misses the mark, due to the reversed hash pointers, as mentioned above. But there is a further difference that this comparison brings to light. In a blockchain, if multiple forks have to be *permanently retained*, consensus across the entire system will be needed for each fork, making it unviable as the number of forks increases. In the CellTree architecture, on the other hand, each individual node is owned and operated by its own crew. Distributing the ownership of nodes to relatively small crews (say, involving a few dozen parties) which can operate in parallel, vastly improves the scalability of the system.

[1] For meaningful guarantees, when a policy rewrites itself, we require the newly resulting policy to validate that the old policy is acceptable to it as a policy to evolve from.

[2] Even architectures like IOTA's Tangle [16], that do not stick to a chain structure use hash pointers in the same direction as blockchains.

We point out that selecting a crew to entirely own a node does raise some issues that need to be handled. Firstly, whenever possible, we would like to ensure that a sufficiently large majority of the parties selected for a crew are honest, and secondly, in the event that the crews for some nodes are compromised (say, because a small crew is used), we would still like to retain as many security guarantees as possible. The first issue is left for a module to address: A typical approach would be to randomly sample from a pool that is considered to have a sufficiently large majority of parties (weighted appropriately by the extent of their computational power, stake in the system etc.). Practical instantiations of such schemes appear in protocols like Algorand [7]. The second issue is addressed by the CellTree architecture, via monitoring: Even if a node's crew is corrupt, any updates to the node's cell will be assimilated into the CellTree only if they are validated and agreed upon by the crews monitoring it. (The exact mechanism of arriving at a consensus across crews is left as a module.)

Dynamic Nodes. Crucial to implementing a Merkle tree based data structure is the ability to dynamically update the contents of previously existing nodes in the tree, so that they can assimilate newly created cells (even if we did not plan to support the evolution of individual cells). But allowing cell data itself to evolve, in a programmable manner, brings a whole new dimension to distributed data repositories. We point out a couple of aspects.

While prior constructions have focused on the immutability or "persistence" guarantee of the blocks in a blockchain, for dynamic data we introduce a notion of *consistency*, to assure that a cell has evolved into its current form in accordance with the policies declared by the cell. These policies govern the modification of the cell's data as well as the policies themselves.

Though persistence is sometimes desirable, many a time it can be a burden on the system. The CellTree design allows individual cells to forget their history (unless they are programmed to retain it – using a suitably programmed cell, a single node could be used to host an entire blockchain).

Excising Malignant Cells. A blockchain, by design, does not allow removing any blocks already accepted to be part of the chain. This (exacerbated by the need to store the entire chain to verify it) creates practical socio-legal complications when illegal data (say, confidential information) is hosted on a blockchain. A CellTree, in contrast, makes it possible to deactivate "malignant" cells, with little impact on the rest of the tree. The node containing a deactivated cell could be brought back to service (with a fresh cell in it), if all the nodes monitoring it cooperate.

3 Background

We shall use a *binary tree* structure to hold the data cells. Below we introduce some basic notation and definitions.

- Each node v in the tree is uniquely identified using a binary string that encodes the path from the root to v. The root node corresponds to the empty string, denoted by ϵ, and for any binary string v, the strings $v0$ and $v1$ denote the left and right children, respectively, of the node denoted by v.
- We say $u \preceq v$ if u is a prefix of v. In terms of the tree, u is an ancestor of v (possibly $u = v$). We write $u \prec v$ if $u \preceq v$ and $u \neq v$. We shall also write $u \prec_\ell v$ (resp., $u \preceq_\ell v$) to denote that the distance between u and v is at most ℓ and $u \prec v$ (resp., $u \preceq v$).
- Given a set of nodes L that form an antichain (i.e., no node in L is an ancestor of another), and a node u, we say that $u \preceq L$ if $\forall v \in L$, $u \preceq v$. We say that $u \precnsim L$ if $\exists v \in L$ s.t. $u \preceq v$. Note that if $v \preceq L$, then the set of nodes $\{u : v \preceq u \precnsim L\}$ forms a subtree rooted at v with L as its set of "terminal" nodes (i.e., nodes with no outgoing edges).

A (full-domain) *hash function* hash takes an input of arbitrary size and maps it deterministically to a fixed size output. We require a *collision resistant* hash function, in which it is infeasible to find two input strings x and y such that $\mathrm{hash}(x) = \mathrm{hash}(y)$. We shall often apply the hash function to a list of values (e.g., $\mathrm{hash}(x, y)$), implicitly requiring that the inputs to hash are first *unambiguously* encoded to allow such lists.

A *Merkle Tree* [12,13] is a technique that allows hashing several blocks of data together into a single block, with the ability to show that a certain data block is part of the hash using a succinct proof [10]. We use a variant of Merkle tree in which the data blocks are associated with all the nodes of a tree (rather than just the leaves). We use the following notation.

- η_v - data block at a node v. (In our construction, η_v itself will be a hash of certain data associated with the node v in a CellTree.) For convenience, we shall require that η_v is defined for all nodes (existing in the tree or not).
- h_v is recursively defined as $h_v := \mathrm{hash}(\eta_v, h_{v0}, h_{v1})$ for nodes that exist in the tree. If a node v does not exist in the tree, $h_v := \mathrm{hash}(\eta_v)$, forming the base of the recursion.
- Given an antichain L, and a node $w \preceq L$, h_w can be computed from the set of values

$$\left(\{\eta_u\}_{w \preceq u \precnsim L}, \{h_u\}_{w \preceq \mathsf{parent}(u) \precnsim L, u \not\precnsim L}\right).$$

We shall also employ a standard digital signature scheme given by a key-generation algorithm and a pair of algorithms (sign, verify). We shall typically use it in the form of a *collective signature* created by a set of parties G (publicly represented by their verification keys in the signature scheme). We write `crewSign.sign`(G, m) to denote the collective signature on a message m produced by G, $\{\text{sign}_g(m)\}_{g \in G}$; similarly, we write `crewSign.verify`(G, m, σ) to denote the verification of a collective signature σ on a message m using the public keys of G. `crewSign.verify` may be programmed with a parameter that specifies how many missing signatures should be tolerated (additional signatures if any, corresponding to public-keys not in G would typically be ignored).

4 Cell Design

Content of each cell consists of two parts, *cell-data* cdata, and a *nucleus* nuc. The nucleus is an entity that typically encodes a fingerprint of the data, and also constrains the trajectory of the data evolution (in the past and the future). The exact functionality of the nucleus is fully customizable and can be used to design a wide variety of cell behaviors, as we explain below.

Typically, the nucleus contains a summary of the cell-data, in addition to some code. The nuclear code consists of algorithms that dictate cell evolution. For an update of a cell evolve from (cdata, nuc) to (cdata′, nuc′) to be valid, both the nuclei involved should *agree* to the evolution. This is required to enable making inferences about the past and future of a cell from the current nucleus (e.g., to ensure that the data stored in a cell is unmodified and unmodifiable). For efficiency purposes, it will be important that the nuclei should arrive at this agreement without seeing the cell data of the other state (past or future) of the cell, but only the nuclear data. Formally, a nucleus is a pair (ncode, ndata), where ncode, consists of 3 algorithms (in some well-defined machine model) and ndata holds data for use in these algorithms. These algorithms are for verifying that the cellular data matches the nucleus of the cell (chkCell), and for checking if the succeeding and preceding nuclei are compatible with the current nucleus (chkNext and chkPrev, respectively). Each of these algorithms takes the nucleus it belongs to (code and data), or the entire cell, as its first argument and outputs $b \in \{\text{true}, \text{false}\}$:

$$\text{chkCell} : \text{cell} \mapsto b$$
$$\text{chkNext} : (\text{nuc}, \text{nuc}^{(+)}) \mapsto b$$
$$\text{chkPrev} : (\text{nuc}, \text{nuc}^{(-)}) \mapsto b$$

Here, $\text{nuc}^{(+)}$ refers to the new nucleus formed after evolution of nuc, and $\text{nuc}^{(-)}$ refers to the previous nucleus that evolves into nuc. For brevity, we shall write chkCell(cell) instead of cell.nuc.ncode.chkCell(cell), chkNext(nuc, $\text{nuc}^{(+)}$) instead of nuc.ncode.chkNext(nuc, $\text{nuc}^{(+)}$), etc.

Initializing an empty cell The nucleus of the empty cell is denoted as nuc_0. We define nuc_0.chkNext to always output true, whereas nuc_0.chkCell and nuc_0.chkPrev always output false. This will prevent the acceptance of the empty cell as the result of evolution of any cell.

4.1 Examples of Nucleus Families

We list a few basic examples of nucleus families below. For concreteness, we illustrate their functions chkPrev, chkNext and chkCell in Figure 2.

- **Static data:** chkPrev ensures that the previous version of the cell had the empty nucleus. chkNext always returns false. This will ensure that a nucleus of this family does not evolve. Then, chkCell ensures that the cell-data cannot change either (assuming collision resistance of the hash function).
- **Blockchain Ledger:** A cell can be used to implement an entire blockchain ledger, which is simply a list of data blocks and the only way in which the cell data can be modified is to append blocks.
- **State Machine:** A nucleus can be used to implement a state machine, so that transitions can occur only according to the edges in a directed graph G over a state space. (The state space can be infinite, as in the case of a counter.) The simple version illustrated in Figure 2 does not use any cdata and there is no additional information in the nuclear data other than the state.

Private Nucleus. Since the nucleus is available outside of the node containing the cell (to the nodes monitoring it), if the contents of the cell are to be protected, the nucleus should retain the cell's secrecy. We point out the possibility that the nuclear data can be kept "encrypted" while still making it possible to run the verification algorithms chkPrev and chkNext. For simple nuclear families this can indeed be achieved efficiently: For instance, for static data or the blockchain ledger above, instead of using the hash of the blocks, a committed hash can be used (e.g., in the random oracle model, appending a random block to the data block before hashing suffices). In the case of the blockchain ledger, we need to also allow virtual datablocks that contain a variable number of blocks (possibly none), so that exactly one virtual block is added during each update, and the actual number of blocks can be hidden.

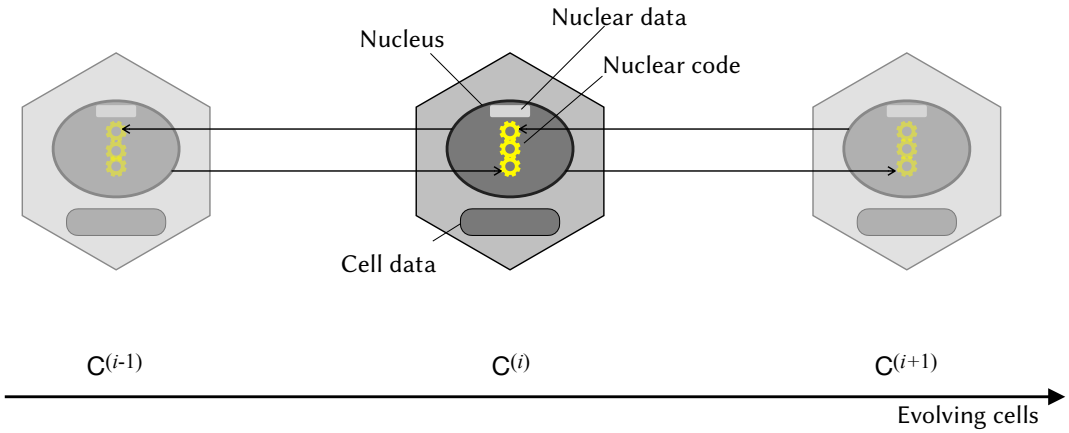

Figure 1 Illustration of cells evolving. The cogwheels indicate the three programs that are part of the nuclear code. The arrows across evolving cells indicate using the next or previous nucleus as inputs to chkNext (top cogwheel) and chkPrev (bottom cogwheel).

For more complex nuclear functionality, (succinct, non-interactive) zero-knowledge proofs can be used to let chkPrev and chkNext verify that the updates are valid without revealing anything else.

External References. The machine model for the nuclear code is implemented using a module `exec`. Apart from standard operations, this machine model may allow references to certain external resources, like current time, cells in other nodes of this CellTree, or even data blocks or code in other blockchains. As a simple example, this allows one to use a timestamp included in a nucleus, and let chkNext accept a new nucleus only if its timestamp is in the past, but later than the timestamp of the current nucleus. Ability to refer to data in other nodes of the CellTree allows a nucleus, for instance, to prove its relation to cells in other nodes (e.g., the nucleus of a cell containing a ledger can prove that a transaction being added to the cell is also present in a global ledger, whose Merkle hash is stored as a cell in a different node).

5 Tree Design

In order to maintain the cells in a single ledger, we shall use a tree structure. Each cell will be exclusively associated with a unique node in the tree, and will be implemented by a set of parties – called the node's *crew*. As time progresses, new cells can be added to the tree and also, each cell may evolve as permitted by its nuclear code.

To make the tree robust to corruption of some cells, we shall have each cell's evolution monitored by several other cells (more specifically, by several cells that are its ancestors in the tree). Also, a cell's evolution is not confirmed until it is *assimilated* into the tree. In this section we describe the mechanisms used for cell evolution, assimilation and access. We begin with the structure and contents of a CellTree, before describing the procedures involved.

Nodes of a CellTree. In a CellTree, the location for a cell is a node in a binary tree. The node address, v, is a binary string representation of the path to the node from the root. For each node v of the tree, there shall be a set of parties G_v — the *crew* operating the node v — who implement the protocols associated with that node, and maintain a copy of the contents of the node. Each member of a crew, $g \in G_v$ has a signing/verification key pair $(\mathrm{SK}_g, \mathrm{VK}_g)$ associated with it, and can be uniquely identified by VK_g (or alternately, a hash thereof). For convenience, we shall denote the collection $\{\mathrm{VK}_g | g \in G_v\}$ by $\widehat{\mathrm{VK}_v}$.

Merkle Multi-Trees. We shall use a variant of a Merkle tree in which the underlying graph structure is that of a *multi-tree* rather than a tree. A multi-tree is a directed acyclic graph such that the set of nodes reachable from any node (via the directed edges) is a tree rooted at the node. While there is a unique tree structure for a CellTree with every node appearing at a fixed location in the tree, the reason we need to use a Merkle Multi-Tree is that the cells in each node can evolve over time, and each version of a cell appears as a separate node in the multi-tree (but has the same address in the binary tree structure).

Recall that in Section 3, we defined a Merkle tree as having data η_u at each node u, and recursively defined a hash pointer for the node $h_u = \mathrm{hash}(\eta_u, h_{u0}, h_{u1})$. But in a Merkle multi-tree, the same binary tree node u is allowed to have different nodes with different values for

Static Data: The nuclear data ndata is simply a hash of the cellular data cdata using a collision-resistant hash function.

$$\mathsf{chkCell}(\mathsf{cell}) = \begin{cases} \mathsf{true} & \text{if } \mathsf{cell.nuc.ndata} = \mathrm{hash}(\mathsf{cell.cdata}) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{chkPrev}(\mathsf{nuc}, \mathsf{nuc}^{(-)}) = \begin{cases} \mathsf{true} & \text{if } \mathsf{nuc}^{(-)} = \mathsf{nuc}_0 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{chkNext}(\mathsf{nuc}, \mathsf{nuc}^{(+)}) = \mathsf{false}$$

Blockchain Ledger: cdata is an array of blocks ($|\mathsf{cdata}|$ refers to the number of blocks in the array and $\mathsf{cdata}[i]$ refers to the i^{th} block). ndata is a triple $(\mathsf{n}, \mathsf{blkHash}, \mathsf{chnHash})$ where n is the number of blocks in the ledger and the other two items are hash values (the last block and the root in a Merkle-*chain*).

$$\mathsf{chkCell}(\mathsf{cell}) = \begin{cases} \mathsf{true} & \text{if } \mathsf{cell.nuc.ndata} = (n, \gamma_n, \mathrm{hash}(\mathrm{hash}((\ldots \mathrm{hash}(\mathrm{hash}(\gamma_1, \gamma_2), \gamma_3), \ldots,), \gamma_{n-1}), \gamma_n)))) \\ & \text{where } n = |\mathsf{cell.cdata}| \text{ and } \gamma_i = \mathrm{hash}(\mathsf{cell.cdata}[i]) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{chkNext}(\mathsf{nuc}, \mathsf{nuc}^{(+)}) = \begin{cases} \mathsf{true} & \text{if } \mathsf{nuc}^{(+)}.\mathsf{ndata.chnHash} = \mathrm{hash}(\mathsf{nuc.ndata.chnHash}, \mathsf{nuc}^{(+)}.\mathsf{ndata.blkHash}), \\ & \quad \mathsf{nuc}^{(+)}.\mathsf{ncode} = \mathsf{nuc.ncode} \text{ and } \mathsf{nuc}^{(+)}.\mathsf{ndata.n} = \mathsf{nuc.ndata.n} + 1 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{chkPrev}(\mathsf{nuc}, \mathsf{nuc}^{(-)}) = \begin{cases} \mathsf{true} & \text{if } \mathsf{nuc}^{(-)} = \mathsf{nuc}_0, \mathsf{nuc.ndata.n} = 1 \text{ and } \mathsf{nuc.ndata.chnHash} = \mathsf{nuc.ndata.blkHash} \\ \mathsf{nuc.chkNext}(\mathsf{nuc}^{(-)}, \mathsf{nuc}) & \text{otherwise} \end{cases}$$

State Machine: A directed graph $G = (V, E)$ over the set of states V is encoded in the algorithms below. ndata is simply a state in V. state_0 refers to the start state of the machine.

$$\mathsf{chkCell}(\mathsf{cell}) = \begin{cases} \mathsf{true} & \text{if } \mathsf{cell.nuc.ndata} \in V \text{ and } \mathsf{cell.cdata} \text{ is empty} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{chkNext}(\mathsf{nuc}, \mathsf{nuc}^{(+)}) = \begin{cases} \mathsf{true} & \text{if } (\mathsf{nuc.ndata}, \mathsf{nuc}^{(+)}.\mathsf{ndata}) \in E \text{ and } \mathsf{nuc}^{(+)}.\mathsf{ncode} = \mathsf{nuc.ncode} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{chkPrev}(\mathsf{nuc}, \mathsf{nuc}^{(-)}) = \begin{cases} \mathsf{true} & \text{if } \mathsf{nuc}^{(-)} = \mathsf{nuc}_0 \text{ and } \mathsf{nuc.ndata} = \mathsf{state}_0 \\ \mathsf{nuc.chkNext}(\mathsf{nuc}^{(-)}, \mathsf{nuc}) & \text{otherwise} \end{cases}$$

Figure 2 Examples of nucleus families. In each case the three algorithms in the nuclear code are described.

η_u, h_{u0}, h_{u1}. We denote such a tuple as mNode_u and it is uniquely addressed by h_u. More precisely,

$$\mathsf{mNode}_u = (\mathsf{nuc}_u, \mathsf{nonce}_u, h_{u0}, h_{u1}). \tag{1}$$

where $(\mathsf{nuc}_u, \mathsf{nonce}_u)$ is the actual data associated with mNode_u, and we let $\eta_u = \mathrm{hash}(\mathsf{nuc}_u, \mathsf{nonce}_u)$.

We shall often refer to a "Merkle subtree" that forms a subgraph of a Merkle multi-tree. The Merkle subtree that is reachable from a node u, and terminated by a set of terminals L is recursively defined as follows.

$$\mathsf{mTree}_L^u = \begin{cases} (\mathsf{mNode}_u, \mathsf{mTree}_L^{u0}, \mathsf{mTree}_L^{u1}) & \text{if } u \precsim L \\ \perp & \text{otherwise} \end{cases}$$

Here, mTree_L^u is uniquely specified by the tuple (u, L, h_u), where h_u is the hash pointer to mNode_u. We let $\mathsf{mTree}_L^u.\mathsf{terminals}$ denote the set of terminals L.

For $u \preceq v \preceq L$, one can *extend* a Merkle subtree rooted at v and terminated at L to one rooted at u and terminated at L, by concatenating it with a Merkle subtree rooted at u and terminated at $\{v\}$. We abbreviate $\mathsf{mTree}_{\{v\}}^u$ as mTree_v^u below. Then, we can denote the above extension as $\mathsf{mTree}_L^u \leftarrow \mathsf{mTree}_v^u || \mathsf{mTree}_L^v$, where the concatenation operator replaces $\mathsf{mTree}_v^v = \perp$ (present in mTree_v^u) with mTree_L^v.

`mEval` denotes the algorithm that, on input mTree_L^u computes h_u or returns an error, if the input is not a valid Merkle subtree (e.g., if the hash pointers included in a node do not match the values computed for the subtrees of that node).

Proof of Assimilation. A proof of assimilation of a node v, with respect to a *root of assimilation* aroot has the following form:

$$\mathsf{poa}_v^{\mathsf{aroot}} = (\mathsf{mTree}_v^{\mathsf{aroot}}, \sigma^{\mathsf{aroot}}), \tag{2}$$

where $\mathsf{mTree}_v^{\mathsf{aroot}}$ is a Merkle tree rooted at aroot with $\{v\}$ as the terminal set, and σ^{aroot} is a signature of the root hash of this tree, signed by the crew of aroot.

Contents of a node. A node stores the following items (accessed via methods of a module `store`).

- The latest version of its cell. (Updated using method `updateCell` and retrieved using `currentCell`.)
- A list of cells that have been propagated rootwards for assimilation (possibly stored in a compressed fashion), with an associated *hash pointer* (stored using the method `addcell`). Any proofs of assimilation received for each cell are also stored (via `addpoa`).
- An ordered list UUlist consisting of nuclei of cell versions since the last rootward propagation. (Items are appended to this list using the method `appendUU`, and the list is retrieved (and emptied) using `flushUU`.)
- A Merkle multi-tree with roots of the form mNode_v, one for each version of the cell that is propagated rootward. (Subtrees of this multi-tree are created and retrieved using methods `addmTree` and `getmTree`).

CellTree Procedures. In Section 5.1 below, we shall specify the following core procedures as part of the CellTree framework:

- **Reading a Cell**: The procedure READ will be invoked by a client to read a cell from a node.
- **Cell Evolution**: The procedure EVOLVE is invoked by the crew of a node to update its cell.
- **Cell Assimilation**: There are two complementary procedures ROOTWARDPROPAGATION and LEAFWARDPROPAGATION that are invoked by the crew of a node to communicate with crews of other nodes, during the assimilation process.

In addition to these procedures, the CellTree framework relies on other procedures (e.g., for scheduling the invocation of various procedures, for creating a new node and assigning it a crew, for deciding what a cell should evolve into, etc.), that are all implemented as modules (see Section 5.2 and Section 6).

Robustness Properties. We define the following desirable properties for a CellTree. These properties are parametrized by a node v, since, even if parts of the tree are malfunctioning, we seek to guarantee these properties to other nodes.

- consistency(v)**:** Let C be the set of cells returned by a set of successful invocations of READ(v) by honest clients. Then, there exists a sequence of nuclei $\mathsf{nuc}^{(0)}, \cdots, \mathsf{nuc}^{(N)}$ such that for all $i \in [1, N]$, both $\mathsf{chkNext}(\mathsf{nuc}^{(i-1)}, \mathsf{nuc}^{(i)})$ and $\mathsf{chkPrev}(\mathsf{nuc}^{(i)}, \mathsf{nuc}^{(i-1)})$ hold and further, for all $\mathsf{cell} \in C$, there exists $i \in [0, N]$, such that $\mathsf{cell.nuc} = \mathsf{nuc}^{(i)}$ and $\mathsf{chkCell}(\mathsf{cell})$ holds.
- correctness(v)**:** Any cell returned by READ(v) to an honest client is equal to a cell assigned to v using the procedure EVOLVE by the crew G_v prior to that (or is the empty cell that every node is initialized with).
- liveness(v)**:** If a cell is assigned to a node v at any point in time (by the crew G_v using the procedure EVOLVE), eventually every invocation of READ(v) by any honest client will return this cell, or a cell assigned to v subsequently.

Note that the above definitions refer to the crew G_v, but the "correct" G_v is not necessarily well-defined if the crews of the CellTree are malicious. A fully formal definition of the above properties requires a formal definition of the correct crew G_v, which we provide in Section 6.1.

5.1 CellTree Procedures

Now we list the core procedures of the CellTree architecture.

5.1.1 Reading a Cell

To read a cell in a node v, a client executes the procedure READ presented below.

```
procedure READ(v, aroot)                                   ▷ aroot ⪯ v
    (G'_v, G'_aroot) ← discover(v, aroot)
    (cell, poa) ← fetch.getCell(G'_v, v, aroot)
    VERIFY(v, aroot, G'_aroot, cell, poa)
    return cell

procedure VERIFY(v, aroot, G_aroot, cell, poa_v^aroot)
    (mTree, σ) ← poa_v^aroot
    h ← hash.mEval(mTree)
    assert mTree.terminals = {v} ∧ mTree.root = aroot
    assert crewSign.verify(G_aroot, h, σ)
    assert cell.nuc = mTree.mNode.nuc
    assert chkCell(cell)
```

The client can use any node $\mathsf{aroot} \preceq v$ as the *assimilation root* where it requires that the cell being read has to be assimilated. The client first discovers the crew for the nodes v and aroot, using a module `discover`. (A typical implementation of this module would require the client to already "know" the crew of one or more nodes

in the CellTree, and would then carry out a search, relying on the fact that crews of a node are expected to know crews of the nodes they monitor and the crews monitoring their node. An example appears in Section 6.)

The algorithm READ also uses a module `fetch` to let a client access the information about a cell from a node's crew (typically, the latest version of the cell for which the crew possesses a proof of assimilation at the specified assimilation root). The client verifies the proof of assimilation (which certifies a nucleus as being assimilated), ensures that the given cell's nucleus matches the one in the proof of assimilation and also verifies that the cell contents are consistent with the nucleus.

5.1.2 Cell Evolution

The cell evolution procedure EVOLVE is invoked at each crew member of a node by the module `selectCell`, to replace the current cell with a new cell $\mathsf{cell}^{(+)}$ that the module has chosen. We expect that `selectCell` achieves a consensus among the crew members before EVOLVE is invoked; on the other hand, the module `store` typically acts locally at each crew member's storage, without requiring any coordination among the crew members.

procedure EVOLVE($v, \mathsf{cell}^{(+)}$)
  $\mathsf{cell} \leftarrow$ `store.currentCell`
  assert $\mathsf{chkNext}(\mathsf{cell.nuc}, \mathsf{cell}^{(+)}.\mathsf{nuc})$
  assert $\mathsf{chkPrev}(\mathsf{cell}^{(+)}.\mathsf{nuc}, \mathsf{cell.nuc})$
  assert $\mathsf{chkCell}(\mathsf{cell}^{(+)})$
  `store.updateCell`$(\mathsf{cell}^{(+)})$
  `store.appendUU`$(\mathsf{cell}^{(+)}.\mathsf{nuc})$

5.1.3 Cell Assimilation

Assimilation is carried out by propagating *assimilation signals* from cells towards the root (Rootward Propagation) and *proofs of assimilation* back from the root to all the cells (Leafward Propagation). This process is designed such that the amount of work a node needs to carry out does not grow as the tree grows. However, we do let the amount of storage required at a node to depend on the *depth of that node*, as a version of the cell is retained at least until all proofs of assimilation for a later version are received.

Rootward Propagation: We describe the procedure invoked by the crew of a node to propagate assimilation signals to its monitoring nodes in the tree. The requisite communication between the node's crew and the crews monitoring it is accomplished through a module `rootward`. This module may use direct communication between the crews of a monitoring and a monitored node, or alternately, use an indirect route (e.g., each node communicating directly only with its parent and children).

procedure ROOTWARDPROPAGATION(v)
  $L \leftarrow$ `rootward.monitored`
  for $b \in \{0,1\}$ **do**
    $\mathsf{mTree}_L^{vb} \leftarrow$ ACCEPTVERIFY(vb, L)
    $h_{vb} \leftarrow$ `hash.mEval`(mTree_L^{vb})
  $\mathsf{UUlist}_v \leftarrow$ `store.flushUU`
  $\mathsf{cell}_v \leftarrow$ `store.currentCell`
  $\mathsf{nonce}_v \leftarrow$ `hash.generateNonce`
  $\mathsf{mNode}_v \leftarrow (\mathsf{cell}_v.\mathsf{nuc}, \mathsf{nonce}_v, h_{v0}, h_{v1})$
  if $v \neq \epsilon$ **then**
    `rootward.send`$(\mathsf{UUlist}_v, \mathsf{mNode}_v)$
  $\mathsf{mTree}_L^v \leftarrow (\mathsf{mNode}_v, \mathsf{mTree}_L^{v0}, \mathsf{mTree}_L^{v1})$
  $h_v \leftarrow$ `hash.eval`$(\mathsf{hash.eval}(\mathsf{cell}_v.\mathsf{nuc}, \mathsf{nonce}_v), h_{v0}, h_{v1})$
  $\sigma_v \leftarrow$ `crewSign.sign`(h_v)
  $\mathsf{poa}_v^v = ((\mathsf{mNode}_v, \perp, \perp), \sigma_v)$
  `leafward.send`(poa_v^v) ▷ send to self
  `store.addcell`(cell_v, h_v)
  `store.addmTree`$(h_v, \mathsf{mTree}_L^v)$
procedure ACCEPTVERIFY(u, L)
  if $u \not\preceq L$ **then**
    return $\perp$
  $(\mathsf{UUlist}_u, \mathsf{mNode}_u) \leftarrow$ `rootward.receive`(u)
  $(\mathsf{nuc}_u, \mathsf{nonce}_u, h_{u0}, h_{u1}) \leftarrow \mathsf{mNode}_u$
  $\mathsf{lastmTree}_L^u \leftarrow$ `store.getmTree`(L, u) ▷ latest version
  $\mathsf{nuc}_u^{(0)} \leftarrow \mathsf{lastmTree}_L^u.\mathsf{mNode.nuc}$
  $\mathsf{nuc}_u^{(1)}, \cdots, \mathsf{nuc}_u^{(t)} \leftarrow \mathsf{UUlist}_u$
  if not CONSISTENT$(\mathsf{nuc}_u^{(0)}, \cdots, \mathsf{nuc}_u^{(t)})$ **then**
    return $\mathsf{lastmTree}_L^u$ ▷ reject update
  for $b \in \{0,1\}$ **do**
    $\mathsf{mTree}_L^{ub} \leftarrow$ ACCEPTVERIFY(ub, L)
    if $ub \precsim L$ **then**
      if `hash.mEval`$(\mathsf{mTree}_L^{ub}) \neq \mathsf{mNode}_u.h_{ub}$ **then**
        return $\mathsf{lastmTree}_L^u$ ▷ reject update
  return $(\mathsf{mNode}_u, \mathsf{mTree}_L^{u0}, \mathsf{mTree}_L^{u1})$
procedure CONSISTENT$(\mathsf{nuc}^{(0)}, \cdots, \mathsf{nuc}^{(t)})$
  for $j \in 1, \cdots, t$ **do**
    $x \leftarrow \mathsf{chkNext}(\mathsf{nuc}^{(j-1)}, \mathsf{nuc}^{(j)})$
    $y \leftarrow \mathsf{chkPrev}(\mathsf{nuc}^{(j)}, \mathsf{nuc}^{(j-1)})$
    if $x = \mathsf{false}$ or $y = \mathsf{false}$ **then**
      return false
  return true

During rootward propagation, an *assimilation signal* from each node u is propagated to every node monitoring it. This message contains the following:

$$(\mathsf{UUlist}_u, \mathsf{mNode}_u)$$

where mNode_u is as defined in Equation 1. We require that nuc_u in mNode_u be the same as the last entry in UUlist_u (stored together).

In the procedure ROOTWARDPROPAGATION, each crew member accepts and verifies a list of nuclear up-

dates from each node it is monitoring. Here, verification involves verifying the consistency between consecutive updates (using chkNext and chkPrev methods in the two nuclei), as well as ensuring that the hash pointers included in an updated node points to valid cells that resulted from verified updates. (In case of verification failure, the previous version of the node is retained.) Then, the crew prepares a node in the Merkle multi-tree, with the current version of its cell, with hash pointers to the verified versions of the cells of its two child nodes. This information is propagated to crews monitoring it, along with a the local list of updates since the last rootward propagation.

Note that the recursive subroutine AcceptVerify could result in the updates of an entire subtree rooted at u being rejected, if the update received from the node u itself fails verification. Further, a faulty update for u could result in updates from its parent (and recursively, all its ancestors till v) being rejected, if the parent of u used a hash pointer that does not match any hash pointer from u. This may seem to suggest that a malicious crew for a single node can disrupt all updates from its ancestors and descendents. However, such attacks can be avoided by well-designed `rootward` and `leafward` modules. In particular, in the simple instantiation in Section 6, the descendents of a node cannot block the nodes updates from being assimilated (though it allows a node from blocking its descendents' updates).

Apart from the module `rootward`, this procedure uses a few other modules: A module `monitoring` is used to obtain the set (subtree) of nodes v monitors: this set is specified in terms of an antichain L, as $\{u | v \prec u \precsim L\}$. The module `store` is used to save the Merkle subtree corresponding to the update that was propagated rootward. The module `crewSign` is used to produce a collective signature that is part of the proof of assimilation of the child nodes' cells. This proof is saved using the module `leafward` and recovered later during leafward propagation.

Leafward Propagation Step: We describe the procedure used by the crew of a node to propagate proof of assimilation from an ancestor node aroot (possibly itself) to one or more of its subtree nodes (see Equation 2). The procedure uses a module `leafward` to collect the proofs and to propagate them to a subtree (after due verification).

The set of nodes for which a node's crew should prepare the proof is also determined by the module `leafward`: The method `pickleaves` returns a set of terminal nodes L and the proof of assimilation $\mathsf{poaset}_L^{\mathsf{aroot}}$ is prepared for all nodes u s.t. $v \preceq u \precsim L$. This set of proofs is transmitted to these nodes using the method `leafward.send`.

---

procedure LeafwardPropagation(v)
- ▷ Process a proof received
- $(\mathsf{aroot}, G_{\mathsf{aroot}}, \mathsf{poa}_v^{\mathsf{aroot}}) \leftarrow$ `leafward.receive`
- $(\mathsf{mTree}_v^{\mathsf{aroot}}, \sigma^{\mathsf{aroot}}) \leftarrow \mathsf{poa}_v^{\mathsf{aroot}}$
- $h_v \leftarrow$ `hash.mEval`$(\mathsf{mTree}_v^{\mathsf{aroot}})$
- assert `crewSign.verify`$(G_{\mathsf{aroot}}, h_v, \sigma^{\mathsf{aroot}})$
- `store.addpoa`$(\mathsf{poa}_v^{\mathsf{aroot}}, h_v)$
- $L \leftarrow$ `leafward.pickleaves`(aroot) ▷ propagate till L
- $\mathsf{mTree}_L^v \leftarrow$ `store.getmTree`(L, v, h_v)
- **if** aroot $\neq v$ **then**
  - $\mathsf{mTree}_L^{\mathsf{aroot}} \leftarrow \mathsf{mTree}_v^{\mathsf{aroot}} || \mathsf{mTree}_L^v$
- $\mathsf{poaset}_L^{\mathsf{aroot}} \leftarrow (\mathsf{mTree}_L^{\mathsf{aroot}}, \sigma^{\mathsf{aroot}})$
- `leafward.send`$(\mathsf{poaset}_L^{\mathsf{aroot}})$

5.2 Modules

A CellTree relies on several modules, but is oblivious to their implementation. In this section, we collect all the modules referred to above, as well as some additional modules that are used by a CellTree deployment. While many of the modules are entirely local to a node's crew, some modules need to coordinate across multiple nodes.

- **Node Creation and Crew Selection.** Creating a new node and selecting a crew for it are tasks carried out collectively by the nodes which would be monitoring the new node, using a module `createNode`. This module could use, e.g., the committee selection protocol used within Algorand [7].
- **Cell Selection.** As mentioned before, cell evolution is triggered by a module `selectCell`, which is responsible for obtaining a consensus among the crew members as to the next version of a cell.
- **Propagation.** As described earlier, the modules `rootward` and `leafward` are used by crews to send and receive assimilation signals and proofs of assimilation. Typically, the `send` and `receive` methods operate in the background and would involve consensus mechanisms and secure communication protocols used within and across crews.
- **Local Storage.** The module `store` is used by each party in a crew to locally store and retrieve various values across separate invocations of the algorithms. Typically, this module requires no communication among crew members (as the consistency guarantees of values being stored are ensured by the other modules).
- **Client Access to a CellTree.** Modules `fetch` and `discover` are used by the procedure Read. These modules may implement access control and and denial-of-service protection.

The proofs of assimilation involve a collective signature by the crew (on a hash value). The protocol for creating such signatures and the algorithm for locally verifying them are encapsulated in the module `crewSign`.

- **Code Execution.** The machine model used to

execute the nuclear code is specified as a module `exec`. The module may support multiple languages and library functions.

• **Hashing.** The hash algorithm and Merkle tree evaluation are implemented by the module `hash`. A method `hash.generateNonce` included in this module can be used to create nonces that control the hash evaluation (e.g., if a time-stamp in the nonce is in the future, the hash evaluation could return an error).

• **Scheduling.** The `sched` module decides when the CellTree procedures are run by the members of a crew.

• **Selection and Evolution of Modules.** The CellTree framework admits different implementations of the various modules to coexist. But the framework does not specify how modules are chosen, and possibly changed over time, instead delegating it to a module `moduleManager` (which governs its own evolution).

6 Instantiating a CellTree: An Example and Analysis

In this section we sketch an instantiation of a CellTree, using a set of relatively simple modules, and give *provable* robustness guarantees for it (in Section 6.1). This instantiation, which we dub $\mathbf{CT_0}$, is only for illustrating the robustness and performance guarantees possible in a CellTree; several improvements are discussed in Section 6.3.

We sketch the design of $\mathbf{CT_0}$ below. This design can be implemented entirely using appropriately defined modules in the CellTree architecture above.

- Every node monitors all its descendent nodes at a distance of at most ℓ, which is a fixed parameter for the entire system.
- During the propagation steps, each node's crew interacts directly only with the crews of its two children and its parent. That is, the crew of a node v has all of the assimilation signals $\{\texttt{rootward.receive}(u)\}_{vb \preceq u}$ (for $b = 0$ or 1) communicated to it by the crew of the node vb; the method `receive` requires that the assimilation signal it receives for a node is signed by the crews of all the nodes monitoring it; for a signature from a crew, a strict majority of the signatures by individual crew members should verify.
  When a node v invokes `leafward.pickleaves` it returns $L = \{v0, v1\}$.
- Only the root node ϵ is used as an assimilation root, and hence proofs of assimilation are not created by the other nodes during rootward propagation.
- The module `discover` implements a simple search algorithm (shown below) to iteratively discover the crews of each node in the path from the root to the node to be read. For simplicity, we assume that the crew of the root node G_ϵ is fixed and hardcoded into the `discover` module used by all nodes (at the time of creating each node). The algorithm is designed to prevent errors in discovery, provided that the root crew is "good" and there is a "good" node in every ℓ long path. Once a crew is discovered, a client can cache it.
- For scheduling the propagation algorithms, we use an "epoch period" of Δ (say half an hour) and a "phase shift" $\Phi = \Delta/t$ for some integer t (say, $\Phi = 5$ minutes) as follows: A node at depth d invokes the rootward propagation algorithm periodically, at time T if $T \equiv d\Phi \pmod{\Delta}$.[3] This allows the assimilation signals from nodes at depth d to cascade to the root, with a latency of at most $\Delta + d\Phi$. The root node invokes the procedure LeafwardPropagation after executing RootwardPropagation; in the other nodes, the procedure LeafwardPropagation is triggered when it receives a proof of assimilation from its parent's `leafward` module.
- We do not fully specify the policy used by the module `createNode` to find the crew for a node being created. But we shall require that it uses a final consensus step to certify the selected crew. A party (which considers itself as monitoring the newly created node) accepts the selected crew if it is certified by the signatures of a strict majority among the *multiset union*[4] of (what the party considers as) the crews of all the nodes that will be monitoring the node being created. This module installs the information regarding the selected crew in all the crews monitoring it, and vice versa.

---

procedure `discover`(v, aroot)
  $G'_\epsilon \leftarrow G_\epsilon$ ▷ G_ϵ is hardcoded
  for all u s.t. $\epsilon \prec u \preceq v$ **do**
    if $\exists G$ s.t. $\{G_u^{(w)} : \epsilon \preceq w \prec_\ell u\} = \{G\}$ **then**
      $G'_u \leftarrow G$
    else
      abort
    for all w s.t. $u \prec w \preceq_\ell u$ **do**
      $G_w^{(u)} \leftarrow \texttt{fetch.getCrew}(G'_u, w)$
  return $(G'_v, G'_{\mathsf{aroot}})$

---

[3] The crew members use roughly synchronized clocks, using a standard protocol like NTP, to determine time. Φ would be chosen to be comfortably larger than the time needed for a crew to carry out the RootwardPropagation procedure (up to `rootward.send`) plus possible delays due to clock skews, and Δ would be chosen large enough to keep the crew's computational effort and network activity low.

[4] In the resulting multiset a party has weight equal to the number of monitoring crews in which it appears. Majority refers to the weighted majority.

6.1 Robustness Analysis

First, we formalize the robustness properties of consistency, correctness and liveness (informally introduced earlier), and present simple assumptions under which these properties are guaranteed by $\mathbf{CT_0}$. Recall that the definitions of robustness properties correctness and liveness referred to *the* crew G_v. To formalize this, we shall first define a notion of a node v being *good*, and then we shall define a crew for v.

To define a good node, we shall refer to the set of parties that a party g *considers as the crew* G_u for a node u (such information is installed in g by the module `createNode`). Note that a party may not consider any set to be the crew of u (if it is not monitoring or being monitored by u's crew, or if the node u has not yet been created).

Definition 1 *For sets of parties G and $\hat{G}$ and a node u, we say that G* considers *$\hat{G}$ as G_u if $\exists H \subseteq G$, with $|H| > |G|/2$ and $\forall g \in H$, g is honest and considers $\hat{G}$ as G_u.*

Definition 2 *We say that a node v is* good *with respect to a set of parties G_ϵ if there exist a sequence of nodes $\epsilon = v_0 \prec v_1 \prec \cdots \prec v_n = v$, and sets of parties $\{\hat{G}_{v_i}\}_{i=0}^{n}$, such that the following hold:*

- $\hat{G}_{v_0} = G_\epsilon$;
- $\forall i \in [0, n]$, $\hat{G}_{v_i}$ *considers itself as* G_{v_i};
- $\forall i \in [0, n)$, $\hat{G}_{v_i}$ *considers* $\hat{G}_{v_{i+1}}$ *as* $G_{v_{i+1}}$.

Note that above, each $\hat{G}_{v_i}$ is required to have an honest majority (as in the definition of *"considers"*). Also, any node being good with respect to G_ϵ, implies that the root node ϵ itself is good.

Recall that the definitions of robustness properties correctness and liveness referred to *the* crew G_v. To formalize these definitions, we shall restrict the definition of these two properties to only good nodes v (with respect to a given G_ϵ), and replace the reference to G_v by (any) crew $\hat{G}_v$ guaranteed by the above definition of v being good.

We shall state our guarantees for the CellTree $\mathbf{CT_0}$ in terms of the following pre-conditions:

Definition 3 *For a node v and a set of parties G_ϵ:*

- $A_v(G_\epsilon)$ *states that the node v is good with respect to G_ϵ.*
- $A_v^i(G_\epsilon)$ *states that in any set of i consecutive nodes in the path from the root node ϵ to v (or if no such set exists, then in the entire path) there is at least one node that is good with respect to G_ϵ.*

We point out that these assumptions are made *per node*, so that it may hold for some nodes and not others. The assumption of a node being good requires only an honest majority among the crew members, and may arguably be made for *every* node in the CellTree, if `createNode` module is implemented is using, say, a scheme like that in Algorand [7], under similar assumptions as used there.

Now we are ready to state the gurantees for $\mathbf{CT_0}$ (a proof sketch is given in Appendix C).

Theorem 1 *In $\mathbf{CT_0}$, let G_ϵ be the root node crew used by the* `discover` *module. Then,*

1. $A_v^\ell(G_\epsilon) \Rightarrow \text{consistency}(v)$,
2. $A_v^\ell(G_\epsilon) \wedge A_v(G_\epsilon) \Rightarrow \text{correctness}(v)$, *and*
3. $A_v^1(G_\epsilon) \Rightarrow \text{liveness}(v)$.

Our guarantees for a node v only depend on (some of) *the nodes in the path from the root to v* being good (even though it may be reasonable to assume that every node is good, by using sufficiently large crews). Not only is this an exponentially small fraction of all the nodes created in the tree (assuming a well-balanced tree), but also it is a set of nodes that are known at the time of creating the node v. Nodes created in the future (even descendents of v) have no effect on any of these guarantees, including liveness.

6.2 Performance Analysis

The CellTree architecture is designed to be scalable, so that a node's communication, computation and storage stay bounded even as the tree grows. The only parameter of the size of the tree that affects a node's complexity is the depth of the node itself (due to the need to store old versions of a cell until newer versions are assimilated at the root), which does not change once the node is created.

Latency: $\mathbf{CT_0}$ does not exploit the multi-level confirmation feature of the CellTree architecture, as it is the root node which issues all proofs of assimilation. As mentioned earlier, an update at a node at a depth d takes at most $\Delta + d\Phi$ time to reach the root, where Δ is the epoch period and Φ is the phase shift parameter. During the leafward propagation step, at each node LeafwardPropagation is triggered right after the parent node has finished their leafward propagation procedure. If the time taken for the procedure to run is at most δ (typically $\delta \ll \Phi$), the proof of assimilation from the root arrives at the node within $\Delta + d(\Phi + \delta)$ time, after an update is made locally at the node.

Storage: Each crew member of a node at depth d stores the following: (1) a single nucleus for each of

the nodes monitored by it, (2) the last cell (of its own node) for which a proof of assimilation was received, all the cells whose nuclei have been propagated rootwards but whose proofs of assimilation have not yet arrived, and the current cell, (3) a single proof of assimilation from the root consisting of $O(d)$ hash values and a single crew-signature, and (4) the nuclei of all the local updates since the last rootward propagation. At steady state, the total storage needed can be bounded as follows. Suppose α is a bound on the nucleus size (plus a nonce and the nuclear hash), and β on the cell size, (typically $\alpha \ll \beta$), r is the rate of updates, and L the latency from above. Then the storage required (apart from a proof of assimilation)

$$(2^\ell + rL)\alpha + (\frac{L}{\Delta} + 1)\beta = d(r\Phi'\alpha + \phi\beta) + c,$$

where $\Phi' = \Phi + \delta \approx \Phi$ and $\phi = \Phi'/\Delta$ is a small constant (e.g., $\phi = 0.2$), and c is independent of the depth. Note that while the storage does depend on the depth, it does so in a moderate way, with the dominant term being $d\phi\beta$.

Communication: A crew member communicates with its peers in the crew, as well as with members of the crews of nodes that monitor it or that it monitors. For concreteness, consider consensus mechanisms which have an initial phase when the actual data is communicated to each party, and a second confirmation stage that uses signatures on hashes of the data; in the optimistic (and typical) case where the confirmation stage accepts the original data, the communication phase need not be repeated. In this case, the total communication incident on each crew member is largely independent of the size of the crews. So we shall focus on the amount of data communicated between *nodes*, rather than the communication incurred by the crew members.

In the assimilation steps, each node communicates directly only with its children and parent nodes. Per epoch, each node receives at most $O(2^\ell r\Delta\alpha)$ bits from the children, and $O(d)$ bits as part of proof of assimilation. The node will also incur communication costs during reads and in other processes it is part of (including `selectCell` for itself and `createNode` for nodes it monitors).

The only potential communication bottleneck in the protocol is in the module `discover`. Crew information of nodes monitored by the root (and nodes close to the root) would be needed by almost all the clients in the system. However, the information that different clients recover from the root (and nodes close to it) is essentially the same, and can be periodically *published* by the root. Then, the `discover` module obtains the crew information of the nodes within (say) 2ℓ distance of the root from information published by the root, and if necessary, carries out a tree traversal starting with a node ℓ away from the root.

6.3 Beyond $\mathbf{CT_0}$

While $\mathbf{CT_0}$ provides reasonable guarantees, it leaves room for much improvement. We discuss several such possibilities below. Mostly these improvements can be implemented by changing the modules.

Improving Liveness. In $\mathbf{CT_0}$, liveness(v) depends on each ancestor node being good (having an honest majority crew). A single bad ancestor can block v's updates from propagating to the root. This can be remedied at the expense of using more complex `rootward` and `leafward` modules, that allow parties to directly communicate with the crews of all the nodes they monitor or which monitor them. This can be used to replace the assumption A_v^1 used by $\mathbf{CT_0}$ for liveness(v) to merely A_v^ℓ.

Removing the Reliance on a Single Root. Another limitation of $\mathbf{CT_0}$ is the reliance on G_ϵ being good. This can be remedied by allowing more nodes in the CellTree to perform the duties of the root: Specifically, many (or all) nodes can issue proofs of assimilation, and the `discover` module can be initialized with the crew information of many such nodes, so that it can choose any of them as its starting point.

Secondly, the assimilation operations carried out by the root's crew (or other nodes close to it) are publicly verifiable (the only secret information used by them being their signing keys, which have corresponding verification keys already published). Further, a set of subtrees can easily "migrate" to a new CellTree. This makes the CellTree less reliant on the root's crew, and thereby would also disincentivize the root's crew from misbehaving.

Modifying the Crews. $\mathbf{CT_0}$ provided no mechanism to modify the crew for any node. In a more realistic instantiation, one would include such a provision (say, by extending the scope of the module `createNode`). This feature would be useful for revoking the public keys of crew members, for replacing misbehaving crew members, or for resizing a crew.

Quality of Service. In $\mathbf{CT_0}$, the propagation schedule, which determines the assimilation rate, is fixed. However, one could support different rates of assimilation to different nodes, thereby providing varying levels of quality of service in terms of efficiency. Further, different nodes could be monitored by different number

of nodes, providing different levels of assurance. These changes can be implemented by appropriately modifying the modules `rootward` and `leafward`.

Other Features. $\mathbf{CT_0}$ does not exploit several features offered by the CellTree architecture, like the ability of modules to evolve, and the ability for multiple modules to coexist in a CellTree. We briefly mention several such features, which are discussed in Appendix B.

- **Pruning, Grafting and Mirroring.** Subtrees can be detached (pruned) from a CellTree, or grafted on to a CellTree – possibly in multiple locations, in case of mirroring – with little effect on the other nodes (which are not monitoring any part of the subtree in question).
- **Excising Cells.** A cell's contents can be deleted, or altered without respecting its program, if *the crews of all the nodes monitoring it* cooperate.
- **Computed Reads.** It is possible to support access to a *function* of a cell's content (with proof).
- **Secret Cells.** The crew members of a node need not be aware of the contents of the cell, but can still provide authorized clients with access to the cell contents, or functions thereof, via secret-sharing or secure multiparty computation protocols.
- **Computing on Multiple Cells.** Concurrent algorithms can be designed to operate on multiple cells, working independently on each cell. This enables maintaining multiple views of a database in different nodes (e.g., in a banking application, each customer's ledger is a partial view of a bank's central ledger).
- **Saplings.** A sapling is a CellTree whose root is assimilated into another parent CellTree. While the parent's crews do not monitor a saplings nodes', they do provide commitment guarantees.
- **Higher Arity Trees.** Higher arity nodes can be easily simulated by allowing the same crew to operate a subtree instead of a single node.
- **Incentivization** The CellTree architecture is agnostic about higher level mechanisms that could be used for incentivizing parties to play the role of crew members. Different parts of the CellTree may employ different incentivization mechanisms.

Finally, we emphasize that a major feature of the CellTree architecture is the ability for different implementations of the same modules to coexist in the tree, offering application specific features in different parts of the tree. Designing such modules and analyzing their effect on the robustness and performance guarantees of a CellTree are left for future research.

7 Related Work

Merkle trees [12,13], and succinct proofs using them [10] have been valuable tools in cryptographers' toolkit for a long time. Cryptographically authenticated blockchains and public ledgers can be traced back to the work of Haber and Scott [8], but became popular only with the advent of Bitcoin [14] and other crypto currencies that used them for recording their transactions. It is outside the scope of this work to survey all the ensuing innovations in this area.

But below we shall discuss a few distributed data repositories which deviate from the blockchain topology, and mention how the CellTree architecture is different from them in its goals and features.

The *Hashgraph* [4] is a distributed ledger with high transaction throughput as compared to blockchains. Its efficient functioning, however, requires that the set of parties involved in the protocol be aware of all the others. (This is comparable to how a single node's crew operates in a CellTree.) The blocks (or events as they are called) in the Hashgraph form a directed acyclic graph (DAG) with hashpointers as edges (unlike in a blockchain, where the graph is a single path), with new blocks pointing to old blocks (like in a blockchain). As in standard blockchains, the selection and confirmation of new blocks in the ledger are probabilistic, but the ability of all the parties to interact with each other allows the use of an efficient voting protocol (rather than one based on, say, proof-of-work). Also, as in standard blockchains, the desired guarantee is that of immutability or persistence of blocks that are confirmed.

The *Tangle* [16] is a permissionless distributed data structure which also uses a DAG structure to store the transactions, again with the goal of increasing the throughput compared to a blockchain. Tangle allows users to be aware of only parts of the entire data structure. Incidentally, the specific algorithms used for building the DAG structure and considering a node confirmed are known to be susceptible to *"parasitic chain attacks,"* and is the subject of ongoing research [6].

The above two systems store data in a graph with hashpointers as edges that has the form of a DAG (rather than a path), to increase the throughput of transactions. The CellTree architecture shares this feature, but promises even better performance when multi-level confirmation can be exploited. Also, the other differences that CellTree has with blockchains continue to apply to these systems as well.

The *Inter-Planetary File System* (IPFS) [5] is a peer-to-peer version controlled file system in which data items, with (optional links to other data items) can be stored. While different in its goals from blockchains, IPFS is also a distributed data repository, with parties

storing some of the IPFS objects in local storage, and accessing others from a peer. The IPFS uses content-addressed sharing, where the address is a hash of the content (with linked objects replaced by their hashes). To detect and avoid duplication, IPFS uses deterministic hashing (no nonces used) so that the same file is hashed to the same address.

Unlike a data repository that is queried using addresses (e.g., the node address in a CellTree), IPFS does not attempt to provide any form of consensus on the "correct" data. All data items, linking to previously existing data items, are valid, and they have their own content-based address. As such, only liveness (all stored data can be retrieved) is of concern to IPFS.

8 Conclusion

In this work we have introduced the CellTree architecture for distributed data repositories. A CellTree is designed to be flexible and heterogenous, allows data and policy evolution, allows parties to focus on only parts of the repository that are of interest to them, and separates out sub-tasks into modules that can be instantiated differently in different parts of the tree. We also illustrated an instantiation of the architecture, $\mathbf{CT_0}$, and presented formal security guarantees.

A CellTree is rich in features and is highly programmable. We leave it for future work to exploit this novel architecture for applications.

References

[1] Enabling blockchain innovations with pegged sidechains. https://blockstream.com/sidechains.pdf, 2014.

[2] Cardano. https://www.cardano.org, 2015.

[3] The bitcoin lightning network: Scalable off-chain instant payments. https://lightning.network/lightning-network-paper.pdf, 2016.

[4] Leemon Baird, Mance Harmon, and Paul Madsen. Hedera: A governing council and public hashgraph network. https://www.hederahashgraph.com/whitepaper, 2017.

[5] Juan Benet. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014.

[6] Andrew Cullen, Pietro Ferraro, Christopher K. King, and Robert Shorten. Distributed ledger technology for iot: Parasite chain attacks. *CoRR*, abs/1904.00996, 2019.

[7] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[8] Stuart Haber and W Scott Stornetta. How to timestamp a digital document. In *CRYPTO*, pages 437–455. Springer, 1990.

[9] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *CRYPTO*, pages 357–388. Springer International Publishing, 2017.

[10] Joe Kilian. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 723–732. ACM, 1992.

[11] Roman Matzutt, Jens Hiller, Martin Henze, Jan Henrik Ziegeldorf, Dirk Müllmann, Oliver Hohlfeld, and Klaus Wehrle. A quantitative analysis of the impact of arbitrary blockchain content on bitcoin. 2018.

[12] Ralph Merkle. Method of providing digital signatures. US Patent 4309569, 1982.

[13] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, pages 369–378, 1987.

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

[15] BBC News. Child abuse images hidden in cryptocurrency blockchain. https://www.bbc.com/news/technology-47130268, 2019.

[16] Serguei Popov. The tangle. http://iotatoken.com/IOTA_Whitepaper.pdf, 2017.

A Modules: Further Details

In this section, we expand on the modules mentioned in Section 5.2.

Node Creation and Crew Selection. A new node is created by the set of nodes who would be monitoring it. They also select a crew for the new node at that time. The exact mechanism used to initiate node creating and carry out crew selection are left to a module, `createNode`. As an example, the crew selection algorithm in this module could be based on the committee selection protocol used within Algorand [7]. We emphasize that, like all modules, multiple implementations of `createNode` can operate in various parts of a CellTree simultaneously.

As part of this module's functionality, the addresses of the members of the crew selected for a newly created node will be made available to the crews of the monitoring nodes. Also the new node's crew is given the addresses of the crew members of nodes monitoring

it. Further, the module is also expected to set (default) modules for the newly created node.

Cell Selection. When a node is created, it has an empty cell, which can then evolve over time. As mentioned before, cell evolution is triggered by a module `selectCell`, which is responsible for obtaining a consensus among the crew members as to the next version of a cell. As an example, the `selectCell` module could use a payment based model to allow a client to choose the next version of a cell, possibly subject to custom policies. Note that irrespective of the mechanism used by `selectCell`, the CellTree algorithm EVOLVE checks that the selected cell is compatible with the existing cell.

Propagation. As described earlier, the modules `rootward` and `leafward` are used by crews to send and receive assimilation signals and proofs of assimilation.

We envisage the `send` and `receive` methods as operating in the background. When they are invoked by a party as part of a propagation algorithm, they may use local buffers to interface with the background processes. The background processes may involve protocols with multiple rounds of interaction among the parties in a crew or across crews. In particular, a typical instance of the method `receive` will have the parties in a crew use a consensus algorithm to agree on the update signals being received, before storing them in a buffer. The `send` and `receive` methods would typically also involve secure (especially, authenticated) communication protocols for crew-to-crew communication.

Note that as part of rootward propagation, there are two services that a node's crew performs: (1) It checks consistency of nuclear updates at the nodes it is monitoring, and (2) It generates proofs of assimilation. One can configure the system so as to make these operations selective, so that not all crews perform all of these operations at every instance of propagation. For instance, the code execution module `exec` (described below) can be configured to require a payment for carrying out consistency checks, or the module `crewSign` can be configured to produce a signature only once every several updates. The `leafward` could be configured to let a crew directly deliver proofs of assimilation from an assimilation root to any of the nodes it monitors (selected via the method `pickleaves`).

Local Storage. The module `store` is used by each party in a crew to locally store and retrieve various values across separate invocations of the algorithms. While the data stored using this module pertains to an entire node, rather than an individual crew member, we envisage this module as operating locally at each crew member location, without any communication. As such, consistency guarantees are derived from the other modules (e.g., `createNode`, `selectCell`, `rootward.receive` and `leafward.receive` all would incorporate consensus mechanisms).

The `store` module may also support *garbage collection* to remove old versions of cells and nuclei of monitored nodes.

Client Access to a CellTree. Recall that as part of the CellTree procedure READ, a client uses a method `getCell` from a module `fetch`, to read a cell from the crew of a node v. Note that for this the client needs to first obtain the addresses of the crew members of the node it is reading from – a functionality encapsulated in the module `discover`. We point out that, within this module, a client will need to know *some* public-key material to bootstrap its access to a CellTree that it is interested in. (This could be in the form of the public-keys of (some of) the crew members for one or more nodes. The module `discover` may itself rely on the module `fetch` to access pieces of information from these nodes.)

The method `fetch.getCell`, along with a cell, also returns a proof of assimilation of the cell, at a specified assimilation root. Note that many versions of a cell may be available at a node, and the module can determine which one to return (possibly after a negotiation between the client and the crew); a natural choice is the latest version which has a proof of assimilation from the specified assimilation root.

The `fetch` module may include a layer of customized access control, to restrict the set of clients who are allowed to query a node. It may also incorporate measures to prevent denial-of-service attacks by malicious clients (e.g., by requiring a proof-of-work by the client).

The proofs of assimilation involve a collective signature by the crew (on a hash value). The protocol for creating such signatures and the algorithm for locally verifying them are encapsulated in the module `crewSign`.

Code Execution. The machine model used to execute the nuclear code is specified as a module `exec`. Making this a module allows different nodes of a CellTree to use different sets of languages. Also, a language can be updated over time, for instance, by introducing new library functions. The `exec` module would use the appropriate interpreter for a nucleus' code (note that we expect that clients reading a node and crews monitoring would use `exec` modules that can correctly interpret its nuclear code).

Hashing. The module `hash` provides methods for evaluating a hash function on a list of inputs (`eval`) and computing the hash-pointer for a Merkle tree (`mEval`).

The module could allow multiple hash algorithms (with a header in the output indicating which one is used), and they could be updated over time. A method `hash.generateNonce` included in this module can be used to create nonces with customized content, to control the hash evaluation (e.g., if a time-stamp in the nonce is in the future, the hash evaluation could return an error).

Scheduling. The `sched` module decides when the CellTree procedures are run by the members of a crew. The frequency with which these procedures are run decides how quickly the changes to a cell take place and are assimilated into a CellTree. The scheduling can be static (run once every epoch), or dynamic (e.g., a leafward propagation message from the parent triggering leafward propagation in child). Scheduling can also react to incentives or negotiations.

Selection and Evolution of Modules. The CellTree framework admits different implementations of the various modules to coexist. But the framework does not specify how modules are chosen, and possibly changed over time. Some modules like `selectCell` are entirely local to a node, and can be freely changed if the node's crew has a consensus on it. But certain other modules like `createNode` involve several crews and will need a consensus from all of them. In either case, we propose that module substitutions follow a mechanism similar to cell evolution, in that the current and next modules must agree with each other for a substitution to be considered acceptable. This would allow long term guarantees regarding the functioning of a node and parts of the tree (e.g., a `createNode` module may not allow its core mechanism to be changed, but only its parameters).

For the sake of completeness, we define a module `moduleManager` that is used to update all the code and data at each node related to the implementation of various modules, including itself. Among other things, this module could be used to implement bug fixes and patches to protocols.

B Additional Features

A CellTree is highly programmable, thanks to the ability to design nuclei with arbitrary code. Apart from this, the CellTree architecture also provides several possibilities that may not be obvious. Here we mention a few of them. We expect that more possibilities will be identified and exploited as CellTrees are used for more applications.

Pruning, Grafting and Mirroring

To prune or detach a node from the tree, the crew of a node can simply stop propagating the assimilation signals from its subtree to the root, and vice versa. Subsequent updates on this subtree are not assimilated into the original tree. Note that outside of this subtree, the original tree can continue to function unaffected by this change, without requiring the crews to take any special action. Also, the pruned out subtree can continue to function as a live CellTree, with the detaching node taking on the role of the root.

Grafting refers to the reverse operation, in which the root of a CellTree is accepted as a newly created node in another CellTree. We note that the proofs of assimilation of a node v at a node u does not depend on the absolute addresses of u and v, but only in the relative address of v with respect to u (implicit in the order in which hashes are accumulated). This allows the root of a grafted subtree to immediately propagate its assimilation signals towards the new root.

Mirroring refers to when a subtree of a CellTree is grafted on to a different location in the CellTree, without detaching itself from its original location. Mirroring creates multiple paths of assimilation and discovery for a subtree, buying it protection against potential failure of one path. A mirrored subtree receives multiple leafward propagations, one from each location it is mirrored at. (The nodes monitoring a node in the newly mirrored subtree would annotate their proofs of assimilation to indicate when they started monitoring it. Such annotations can be included in the nonces.)

Excising Cells

A cell may be programmed to disallow removing (or even altering) its data. Even so, if the crew of a node wishes to remove (or alter) the contents of its cell from the CellTree, it may do so *in cooperation with the crews of all the nodes monitoring it.* Such a feature, combined with the ability to change the crew of a node, is helpful in removing any illegal content discovered in a node.

Computed Reads

While the method `fetch.getCell` would typically be used to retrieve an entire cell, it could offer a partial view of the cell, provided that the nucleus' chkCell algorithm can verify this partial view. For instance, if a cell contains a blockchain ledger, the partial view may include only the last block, and an accumulated hash of the previous blocks. More generally, chkCell could accept a zero-knowledge proof that a given function of the cell is consistent with a commitment (hash) of the cell that is included in the nucleus.

Secret Cells

Using cryptographic techniques, it is possible for a crew to maintain a cell without any crew member (or even a collusion of malicious parties in the crew) learning the contents of the cell. Authorized clients can access this cell data — or a function thereof (as a computed read) — via secure computation protocols. A cell could also be used to pool secret data from a set of clients, and allow authorized clients to query the resulting data.

Computing on Multiple Cells

Evolution of a cell can depend not only on its own data, but also on the cells in other nodes, by using appropriately designed nuclei, using an `exec` module that allows reference to cells in other nodes. Dereferencing would involve reading a cell (subject to access controls). Such computations would be relevant in maintaining multiple views of a database in various nodes (e.g., in a banking application, each customer's ledger is a partial view of a bank's central ledger).

We remark that since the nodes are updated individually and asynchronously, the nuclei need to be carefully designed, so that the consistency guarantee of individual nuclei translates to a guarantee that the global updates were carried out correctly.[5]

Saplings

A sapling is a CellTree "supported by" another CellTree for time-stamping purposes. The sapling will be a separate CellTree whose root's nucleus is included in the nucleus of a node in the parent tree (along with additional information like time-stamps). Saplings can be used to create "enterprise subtrees," whose contents are entirely hidden from the rest of the tree, but with *commitment guarantees* provided by the larger tree.

Higher Arity Nodes

Since the depth of a node in the tree affects its performance guarantees, restricting to a binary tree may appear sub-optimal. However, higher arity nodes can be easily simulated by allowing the same crew to operate a subtree instead of a single node. For instance, a crew which operates a node v and its two children $v0$ and $v1$, can simulate a node v with four children $(v00, v01, v10, v11)$ but at a smaller depth.

[5] As an illustration, suppose we wish to support swapping the cells in nodes u and v (say C_1 and C_2, respectively). First the cell in u is updated from C_1 to (C_1, C_2) (with its nucleus carrying hashes of both the cells); during the time this updated cell is assimilated by the nodes monitoring u, the cell C_2 in v should not change, so that chkNext accepts the update. Later the node v is updated to replace C_2 with C_1; to ensure that this update is part of a swap, chkNext will confirm that the cell in u has contents (C_1, C_2), as expected after the above step. Finally, u will be updated again, so that it replaces (C_1, C_2) with C_2; chkNext will confirm that v has C_1 at this point.

Incentivization

The CellTree architecture is agnostic about higher level mechanisms that could be used for incentivizing parties to play the role of crew members. In particular, clients wishing to load their data into a node's cell could be required to make a payment (using digital currencies or legal tenders) to the parties in that node's crew; in turn, the crew members could be required to make payments to the nodes monitoring them, for their continued services.

C Proving Robustness of $\mathbf{CT_0}$

Here we sketch a proof for Theorem 1.

Proof sketch: Firstly, we argue that under the assumption $A_v^\ell(G_\epsilon)$, `discover`(v) may only return a crew that is *considered* to be G_v by a good node u monitoring v (or it will abort). Consider the set of ℓ ancestors of v all of which monitor v. By assumption $A_v^\ell(G_\epsilon)$, there is a node u among them which is good w.r.t. G_ϵ. Then, there exist a sequence of nodes $\epsilon = u_0 \prec u_1 \prec \cdots \prec u_n = u$, and sets of parties $\{\hat{G}_{u_i}\}_{i=0}^n$, such that $\hat{G}_{u_0} = G_\epsilon$, and the successive crews consider the next one in the sequence to be the correct crew. Hence if `discover` queries $\hat{G}_{u_i}$ for the crew of u_{i+1}, it will receive $\hat{G}_{u_{i+1}}$ in response. Then, inductively, `discover` will accept only $\hat{G}_{u_i}$ as the crew for u_i (base case being $u_0 = \epsilon$ (or will abort). Then, it will return a crew only if $\hat{G}_{u_n}$ considers it to be G_v.

Next, we argue that if a proof of assimilation obtained by Read(v) verifies, then the cell retrieved along with it must have a nucleus that was verified by a good node u monitoring v. Again, consider the sequence of nodes $\epsilon = u_0 \prec u_1 \prec \cdots \prec u_n = u$ from above. The root's crew signed the proof of assimilation only because all the assimilation signals that were assimilated were signed by all the nodes monitoring the respective nodes. In particular, u_1 which is monitored by $\epsilon = u_0$ must have signed the assimilation signals that the root incorporated in the proof. In turn, u_1 required a signature by u_2 on the assimilation signature that u_1 incorporated into the signal sent to u_0. Continuing in this manner, we are guaranteed that the proof of assimilation signed by the root corresponds to an assimilation signal sent by $u_n = u$. Thus any cell returned by Read(v) will have a nucleus that was assimilated by the node u. u, being a good node that monitors v, would incorporate a

nucleus of v into this signature only after a consistency check.

Now we are ready to prove consistency. For this, we consider the sequence of nuclei that u has maintained for v, $\mathsf{nuc}^{(0)}, \cdots, \mathsf{nuc}^{(N)}$ so far. By the above argument, the nuclei of any cell cell accepted by READ appears in this sequence (i.e., $\mathsf{cell.nuc} = \mathsf{nuc}^{(i)}$ for some i). Further, u's consistency checks ensure the conditions $\mathsf{chkNext}(\mathsf{nuc}^{(i-1)}, \mathsf{nuc}^{(i)})$ and $\mathsf{chkPrev}(\mathsf{nuc}^{(i)}, \mathsf{nuc}^{(i-1)})$. Also, the verification in READ ensures that $\mathsf{chkCell}(\mathsf{cell})$ holds. These are precisely the conditions required by the consistency guarantee.

The correctness guarantee follows from consistency and the extra condition of $A_v(G_\epsilon)$, as a good node for v will propagate only the updates created by EVOLVE for assimilation.

Finally, if $A^1_v(G_\epsilon)$ holds we argue that liveness holds. A non-trivial issue here is to ensure that a node's siblings (or cousins) or descendents should not be able to prevent its rootward propagation. This is ensured in the procedure ROOTWARDPROPAGATION, by retaining a valid update from one child, even if the update from its sibling is discarded for failing consistency checks. Also note that if there is a path of all good nodes from ϵ to v, the proof of assimilation is routed back to v's crew. Finally, we also rely on the fact that `discover`(v) will never abort and the procedure READ will also succeed, assuming $A^1_v(G_\epsilon)$. □