Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

A Formal Analysis Of Hashgraph And Its Accountability Properties

Marcel Flinspach

Course of Study: Informatik

Examiner: Prof. Dr. Ralf Küsters

Supervisor: Dipl.-Math. Mike Graf

Commenced: January 11, 2022

Completed: July 11, 2022

Abstract

The *Hashgraph algorithm* is a distributed ledger technology (DLT) consensus algorithm that is an alternative to conventional blockchains. Generally, a distributed ledger can be seen as a database of transactions that is replicated across serveral locations, typically run by multiple parties. In order to reach an agreement on the validity and order of transactions, DLTs typically rely on consensus protocols as a key component.

Participants of the Hashgraph algorithm locally manage a hashgraph. This is a directed acyclic graph of events. All events include, among other (meta)data, mainly transactions that were submitted by clients. In order to reach a consens, Hashgraph utilizes so-called *virtual voting* so that parties with different hashgraphs assign all events the same position in the total order of events. We call this desirable property *consistency*, which allows different participants to calculate and agree on the same order of transactions.

Accountability is a well-known concept in distributed systems and cryptography but new to blockchains and DLTs in general. With this concept, misbehaving parties violating predefined security goals can be identified and held accountable with undeniable cryptographic evidence to incentivize participants to behave honestly.

In this work, we put forward a rigorous proof that Hashgraph does achieve accountability w.r.t. consistency. That is, participants that misbehave by calculating a different order of transactions, by not following the Hashgraph protocol, can always be identified and rightfully blamed. To achieve this, we construct an iUC model of the hashgraph protocol with the necessary additions to hold dishonest participants accountable. In particular, we prove under relatively mild assumptions that honest participants, following the Hashgraph algorithm, will always assign events in their hashgraph the same order. That is, honest participants can reach a consens on the total order of events and transactions. Due to the real-world applications of Hashgraph, we believe this result is of independent interest.

Contents

List of Figures

1 Introduction

The invention of the blockchain in 2008 with Bitcoin [15], a payment system solution without central authority, instigated interest in blockchain technologies and distributed ledger technology (DLT) in general. The focus of interest and research on blockchains and distributed ledgers has developed byond the original concepts of alternative payment systems. Specifically in the financial sector, a plethora of use cases emerged for blockchain and DLT solutions with Corda [5] and Hyperledger Fabric [1] being among the most widely adopted distributed ledgers.

DLTs. In essence, a distributed ledger can be seen as a synchronized, replicated database across multiple locations, generally run by different mutually non-trusting participants, managing a ledger of transactions. Each participant in the network usually preserves its own copy of the ledger. DLT offers new ways to validate and order transactions across distributed systems: Unlike traditional approaches which typically require central institutions and their applications to carry out transactions (e.g. money transfers), DLT is intended to reduce reliance on central parties. In order to reach an agreement on the validity and order of transactions in the distributed ledger, DLTs typically rely on so-called consensus protocols.

Blockchains. A blockchain is a distributed ledger with a growing list of records, called blocks, that are linked together to from a chain of blocks by means of cryptographic hashes. Except for the first created block, each block in the chain is linked to its preceding block by including a cryptographic hash of this block. The blocks usually contain more metadata (e.g. a timestamp of block creation) and transactions. Participants of the blockchain network usually store the chain of blocks, called blockchain, locally.

Depending on the blockchain, no special authorization or identification is required to join the blockchain network. Such blockchains (e.g. Bitcoin) are referred to as *public* or *permissionless*. In these networks, a consens among participants is usually reached by solving a computationally complex mathematical puzzle that requires intense computational effort. This concept is known as *proof of work*, which is widely used for many permissionless blockchains (e.g. Bitcoin and Ethereum [18]). This class of consensus algorithms often suffer from enormous energy consumption, slow transaction speeds, high transaction fees, and lack of finality. Therefore, proof of work consensus algorithms are for many application purposes not sufficient. Analogous to public blockchains, there are *private* or *permissioned* blockchains that allow access for authorized participants only.

Hashgraph. Since first introduced in 2016 [2], the Hashgraph protocol [2, 3, 4, 16] gained a lot of attention [9, 17]. In January 2022, the Hashgraph algorithm was made open source under the Apache License [13]. The most prominent implementation of the Hashgraph algorithm is the Hedera Hashgraph distributed ledger that is owned and managed by the Hedera Governing Council [3]. The council's members encompass many important companies, including Google, IBM, Boeing, Deutsche Telekom, and LG [12].

The Hashgraph consensus protocol solves many issues that exist with popular distributed ledgers; namely, high transaction throughput, high efficiecy, low cost, fairness, ACID compliance, DoS resistantance, and Byzantine fault tolerance [3, 4].

Accountability. Accountability is a well-known concept in distributed systems and cryptography; however, there are not many applications in the domain of distributed ledgers and blockchains. Graf et al. proposed in [10] a formal treatment of accountability for distributed ledgers. Moreover, the authors of this paper proved that Hyperledger Fabric, with some additional changes, does achieve accountability w.r.t. consistency.

Related Work. The Hashgraph consensus protocol was proved in [2] to be consistent. That is, honest participants, following the Hashgraph algorithm, do calculate the same order of transactions under the assumption of a supermajority (i.e., more than $\frac{2}{3}$) of honest participants. However, the proof in [2] is incomplete in several key portions; thus, a unrestrictedly convicing security analysis of Hashgraph does not exist, yet. Besides the correctness proof of Karl Crary in [8] using the Coq proof assistant, we are not aware of any other work in this field.

Our approach of proving accountability w.r.t. consistency generally follows the proof of accountability w.r.t. Fabric* in [10]. Particularly, we adopt many terms and general concepts presented in [10].

Contribution. In this work, we will apply the accountability framework from Küsters et al. in [14] to proof our main result that the Hashgraph protocol satisfies accountability w.r.t. consistency. To do so, we first formally define accountability w.r.t. consistency for Hashgraph. We will argue that fork-freeness is an indispensable security property to show accountability w.r.t. consistency without assuming a supermajority of honest participants.

Furthermore, we will demonstrate in a detailed proof that Hashgraph is consistent. By this, we aim to close the gaps of the proofs in [2] that are relevant to consistency.

Structure of this work. We first introduce the Hashgrap protocol in Chapter 2. In Chapter 3, we present our Hashgraph model after we provide a brief introduction to the iUC framework that we use for our model. The accountability framework we use is explained in Chapter 4. Finally, in Chapter 5 we present all security notions, formally define accountability w.r.t. consistency (and fork-freeness), and present all proofs, relevant to Hashgraph.

2 The Hashgraph Consensus Algorithm

We explain the Hashgraph algorithm as presented in [2], but we also adopt some notations from [8].

2.1 Overview of the Hashgraph Protocol

In Hashgraph, there are *clients* and *nodes*. Clients submit transactions, arbitrary messages, to nodes. The consensus on the order of transactions is only achieved by nodes which belong to the same Hashgraph session. All nodes in a session are predefined in the set

$$\mathsf{nodes} = \{pid_1, \dots, pid_n\}$$

with arbitrary but fixed party IDs pid_i.[1] We define $n := \mathsf{nodes}$ to be the fixed number of nodes in one session. We later prove that all honest nodes calculate a prefix of the list of ordered transactions, as long as there is a supermajority (defined to be more than $\frac{2}{3}n$) of nodes that behave honestly.

2.1.1 Events and Hashgraphs

Instead of ordering transactions directly, nodes group transactions into so-called events. Then, nodes can reach a consensus on the total order of events, and extract subsequently the order of transactions from ordered events. Events are created by nodes and form the vertices of the hashgraph, a directed acyclic graph, that each nodes maintains locally. Formally an event is a 7-tuple,

$$(pid, \mathit{eventID}, \mathit{selfParent}, \mathit{otherParent}, \mathit{txs}, \mathit{ts}, \sigma),$$

where

- pid is the party ID of the node that created this event,
- $\mathit{eventID}$ is its ID which is computed as hash over $(\mathit{selfParent}, \mathit{otherParent}, \mathit{txs}, \mathit{ts})$,
- $\mathit{selfParent}$ is the eventID of an event created by pid,
- $\mathit{otherParent}$ is the eventID of an event created by some node other than pid,
- txs is a set of transactions,
- $\mathit{ts} \in \mathbb{N}$ is the time pid created this event,
- σ is the signature of the event signed by pid over $(\mathit{selfParent}, \mathit{otherParent}, \mathit{txs}, \mathit{ts})$.

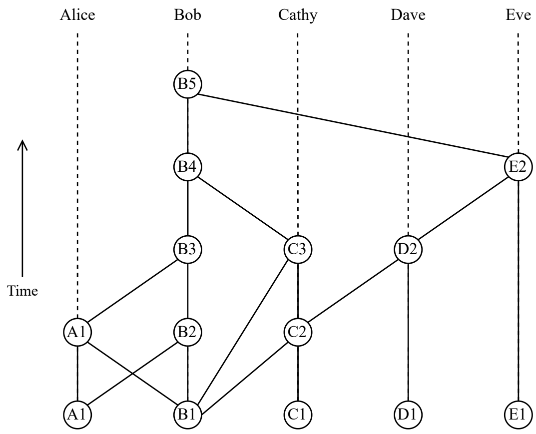

Figure 2.1: A hashgraph

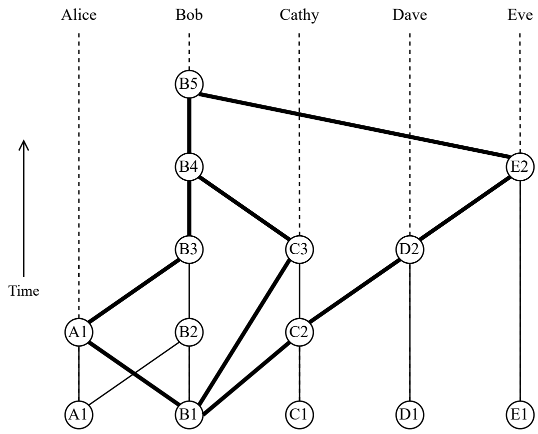

Figure 2.2: A hashgraph (paths highlighted)

The edges of a vertex are therefore defined by the event IDs $selfParent$ and $otherParent$ (cf. Figure 2.1). We denote the hashgraph of a participant pid as $G(pid)$ or simple G. Formally, a hashgraph is a set of events with the addition of the genesis event $(\perp, \perp, \perp, \perp, \perp, \perp, \perp)$ that is contained in all hashgraphs of honest nodes.

2.1.2 Hashgraph Terminology

We first define some general concepts of the Hashgraph algorithm. Let G be the hashgraph of some node pid and $E, E' \in G$ be two events.

- Event E is an *ancestor* of E' if $E = E'$ or there exists a down-path from E' to E along the vertices of G.
- E is an *self-ancestor* of E' if $E = E'$ or there exists a down-path from E' to E such that each event included in this path was created by pid.
- The pair of events (E, E') is a fork if E and E' have the same creator, but neither is a self-ancestor of the other.
- E can *see* E' if E' is an ancestor of E, and the ancestors of E do not include a fork by the creator of E'.
- E can *strongly see* E' if E can see E' and there is a set $S \subseteq G$ of events such that $|S| > \frac{2}{3}n$, all events in S have pairwise different creators, and for all $E_s \in S$ applies E sees E_s and E_s sees E'.

Example 1. Figure 2.2 illustrates the concept of strongly seeing; in this hashgraph with $n = 5$, event B5 can strongly see event B1: Let $S = \{$A1, C3, D2, E2$\}$, then $|S| = 4 > \frac{2}{3} \cdot 5$, B5 can see all events in S, and all events in S can see B_1.

[1]Throughout this work, we usually use the terms party, participant, and node synonymously if it is clear that we refer to an actual node participant in a Hashgraph session.

2.1.3 Synchronization of hashgraphs

In this section, we assume all nodes are honest and follow the described procedures.[2] During a run of a hashgraph session, all nodes try to sync infinitely often with other random nodes. A sync is a procedure where one node, the initiator of the sync, sends a local copy of its hashgraph to another node pid. In Hashgraph, this process is also called gossiping.

Upon receiving the synced hashgraph, the pid merges all events with valid signatures and hashes into its local one. Additionally, the receiving node creates a new event E with self-parent being the last event it created or the genesis event, in case it has not created any events yet. The other-parent is set to the last event, say E', the initiator node created; that is, there is no other event in the gossiped hashgraph with self-parent being the event ID of E'. The receiving node includes all transactions, submitted by clients that have not yet been included in any other event, into the set txs of event E.

Lastly, pid runs the following three functions in the given order: *1)* `divideRounds`, *2)* `decideFame`, and *3)* `findOrder`. The three functions above are abbreviated with $\mathcal{H}_{alg}$. The output of $\mathcal{H}_{alg}$ with the local hashgraph of pid as input is a total order of events. In the following, we discuss all three functions in more detail.

1) The `divideRounds` functions (cf. Figure 2.3) assigns all events in the local hashgraph a *round created* (or simple *round*) number. Initially, the genesis event is assigned round 1. All other events have a round number r that is equal the maximum round of its two parent events. However, there is one exception: If an event E can strongly see more than $\frac{2}{3}n$ round r witnesses, it will be assigned round number $r + 1$ instead. In this case, E is a *witness* event. A witness event is the first event created by a member in round.

   We denote with $\mathcal{W}_r^{pid}$ the witness events of round r in the local hashgraph of pid. We usually omit the participant and just write $\mathcal{W}_r$ when it is clear to which hashgraph this set belongs.

2) The `decideFame` function (cf. Figure 2.4) decides for all witness events $E \in G$ whether it is famous or not. This is done by *virtual voting*, where a supermajority of events from different participants decide whether an event is *famous* or not. Each participants runs this algorithm locally, with their own local hashgraph as input, with no additional network communication.

   Only witness events are eligible to participate in the election. Let E_r be a witness event in G of round r. During the first voting round, all witnesses $E_{r+1} \in \mathcal{W}_{r+1}$ vote for E_r. A witness votes $\beta = \texttt{true}$ for E_r's fame if it can see it. Otherwise, it will vote $\beta = \texttt{false}$ and thus against it. This concludes the first voting round.

   Subsequent voting rounds $i \geq 1$ are different compared to the first round. Let $E_{r+i+1} \in \mathcal{W}_{r+i+1}$ be a round $r + i + 1$ witness. Let $\mathcal{W}$ be the set of all round $r + i$ witnesses in G that E_{r+i+1} can strongly see, and let v be the majority vote of events in $\mathcal{W}$ for E_r. During virtual voting, there are normal voting rounds and so-called *coin rounds*. During a normal voting round, E_{r+i+1} always vote for E_r w.r.t. the majority decision v. If there is a supermajority of events in $\mathcal{W}$ with the same vote for E_r, then E_{r+i+1} will *decide* the fame for E_r according to the supermajority decision v. We denote this voting result with $\mathsf{decision}_G(E_r, E_{r+i+1}, v)$. At this point, the virtual voting for E_r is ended. Now, suppose we are in a coin round. In coin rounds

[2]The handling of exceptions due to dishonest nodes is discussed in the subsequent chapters

events cannot decide the famousness for events. However, events still vote for E_r. If there is a supermajority of round $r + i$ events in $\mathcal{W}$ that agree on vote v, E_{r+i+1} will still vote with v for E_r. Otherwise, E_{r+i+1} "flips a coin" for the vote E_r by voting according to its last bit of the signature.

For later proofs we will need the predicate

$$\mathsf{vote}_G(E_r, E_{r+i+1}, \beta)$$

that is true if event E_{r+i+1}, in voting round i, votes for event E_r's fame with β.

3) The `decideFame` function (cf. Figure 2.5) assigns events a position in the total order of events. First, we need to define the following definition: A Unique famous witness is a famous witness that does not have the same creator as any other famous witness created in the same round. Notice that in the absence of forking, each famous witness is also a unique famous witness.

Let $E_0 \in G$ be an arbitrary event. At the beginning, `decideFame` determines if there can even exist an ordering of E_0 at this point, by checking if the *round received* number of E_0 exists. The round received number r of an event E_0 is defined to be the first round where all unique famous witnesses of round r are descendants of E_0. If E_0 has a round received number, `decideFame` calculates the set $\mathcal{S} \subseteq G$ for E_0, which is defined as

$$\begin{aligned}\mathcal{S} = \{E_1 \in G \mid \exists E_2 \in G \colon\ & E_2 \in \mathcal{U}_{r_r} \\ & \wedge E_1 \textit{ is a self-ancestor of } E_2 \\ & \wedge E_0 \textit{ is an ancestor of } E_1. \\ & \wedge E_0 \textit{ is not an ancestor of } E_1\textit{'s self-parent}\}.\end{aligned}$$

Now, the *consensus timestamp* of event E_0 is determined by taking the median of the timestamps of the events in S.

Lastly, the complete order of events is calculated: Events are sorted in ascending order by 1. the round received number, 2. remaining ties by consensus timestamps, and 3. any remaining ties lexicographically by their signature.

We later proof that hashgraph is consistent, i.e., honest participants calculate the same order of events if there are less than $\frac{1}{3}n$ dishonest participants.

```
function divideRounds() :                                   {Assign rounds to events and mark witnesses.
    for all event x ∈ G(pid), in topological order do:      {Iterate over events in the hashgraph of pid.
        r ← 1.
        if event x has parents:
            r ← max round of parents of x.
        x.round ← r.
        if x strongly sees more than 2/3 n round r witnesses:   {If event x is a witness.
            x.round ← x.round + 1.
        x.witness ← (x has no self parent) ∨ (x.round > x.selfParent.round).
```

Figure 2.3: The `divideRounds` function of hashgraph.

```
function decideFame() :                                                    {Decide which events are famous.
    for all event x ∈ G(pid), in order from earlier rounds to later do:
        x.famous ←?.                                                       {Fame of x is undecided at first.
        if ∃r ∈ ℕ s.t. x ∈ W_r^pid:                                        {If x is a witness in round r.
            for all y ∈ W_{r+1}^pid do:                                    {First round of the election; i = 0.
                y.vote ← y can see x?.
            for i = 1, . . . s.t. W_{r+i+1}^pid ≠ ∅ do:                    {Further rounds of the election; continues
                for all y ∈ W_{r+i+1}^pid do:                              until there are no more witnesses.
                    W ← {w ∈ W_{r+i}^pid | y strongly sees w}.
                    v ← majority vote in W(true for tie).
                    t ← number of events in W with a vote of v.
                    if i mod c > 0:                                        {This is a normal round.
                        y.vote ← v.
                        if t > 2/3 n:                                      {If supermajority is reached, decide fame of x.
                            x.famous ← v.
                            break out of i-loop.
                    else:                                                  {Begin coin round.
                        if t > 2/3 n:
                            y.vote ← v.
                        else:
                            y.vote ← last bit of y.signature.              {Event y votes pseudorandomly.
```

Figure 2.4: The `decideFame` function of hashgraph.

```
function findOrder() :                                      {Establish total order of events and transactions.
    for all event x ∈ G(pid) do:                            {Check and calculate the roundReceived
        if there exists a round r ∈ ℕ  s.t.                  number of x, if it exists.
                1. there is no event y with y.round ≤ r that has
                    (i) y.witness = true and
                    (ii) y.famous = ?; and
                2. x is an ancestor of every unique famous witness in round r; and
                3. r is minimal, i.e., conditions 1. and 2. are false for all r′ < r :
            x.roundReceived ← r.
            S ← set off all events z ∈ G(pid), where
                    1. z is a self-ancestor of a round r unique famous witness, and
                    2. x is an ancestor of z, and
                    3. x is not an ancestor of z's selfparent.
            x.consensusTimestamp ← median of the timestamps of all events in S.
    E ← list of all events in G(pid) with a round received number.
    E ← sort events in G(pid) in ascending order by
            1. roundReceived number,
            2. consensusTimestamp, and
            3. lexicographically by their signature in case of ties.
    return E.
```

Figure 2.5: The `findOrder` function of hashgraph.

3 Security Model of Hashgraph

3.1 Introduction to the iUC Framework

The iUC framework [6] is a highly expressive universal composability model with many sensitive defaults for easier use. In particular, the framework offers a convenient template for specifying various prtocols and systems. We begin reiterating the general structure of protocols as described in [6].

A protocol $\mathcal{P}$ in the iUC framework is specified via a system of machines $\{M_1, \ldots, M_l\}$. Each machine M_i implements one or more roles of the protocol, where a role describes a piece of code that performs a specific task. In a run of a protocol, there can be several instances of every machine, interacting with each other (and the environment) via I/O interfaces and interacting with the adversary (and possibly the environment) via network interfaces. An instance of a machine manages one or more so-called *entities*. An entity is identified by a tuple $(pid, sid, role)$ and describes a specific party with party ID (PID) pid running in a session with session ID (SID) sid and executing some code defined by the role $role$ where this role has to be (one of) the role(s) of M_i according to the specification of M_i. Entities can send messages to and receive messages from other entities and the adversary using the `I/O` and network interfaces of their respective machine instances.

3.1.1 Structure of Protocols

Roles. Roles are piece of codes that perform a specific task in a protocol $\mathcal{P}$. Each role in $\mathcal{P}$ is implemented by a single machine M_i; however, one machine can implement multiple roles. Roles of a protocol can be either public or private. `I/O` interfaces of public roles are accessible by other entities belonging to roles in the same protocol $\mathcal{P}$ or the environment/unknown higher level protocols. Whereas `I/O` interfaces of private roles are only accessible by other entities belonging to the same protocol. For a protocol $\mathcal{P}$ with p public and q private roles, we write $(\mathtt{pubrole}_1, \ldots, \mathtt{pubrole}_p \mid \mathtt{privrole}_1, \ldots, \mathtt{privrole}_q)$. If two protocols are combined to a new protocol, all private roles will remain private, whereas previously public roles can be either public or private.

Subroutines. A machine M_i implementing a specific role can implement other roles as subroutines. Then, the `I/O` interfaces of M_i will be connected to the `I/O` interfaces of the machine implementing the (as subroutine) specified role.

Setup for the protocol $\mathcal{Q} = \{M_1, \ldots, M_n\}$:

Participating roles: list of all n sets of roles participating in this protocol; each set corresponds to one machine M_i.
Corruption model: corruption model of $\mathcal{Q}$ (e.g., incorruptible or dynamic corruption).
Protocol parameters*: e.g., externally provided algorithms or variables parametrizing a machine.

Implementation of M_i for each set of roles:

Implemented role(s): the set of roles implemented by this machine.
Subroutines*: a list of all (other) roles that this machine uses as subroutines.
Internal state*: internal state variables of M_i that are used to store data across different machine invocations.
CheckID*: algorithm for deciding whether this machine is responsible for an entity $(pid, sid, role)$.
Corruption behavior*: description of the **DetermineCorrStatus** and **AllowAdvMessage** algorithms.
Initialization*: this block is executed the first time an instance of the machine M_i accepts a message; useful to, e.g., assign initial values that are globally used for all entities managed by this instance.
EntityInitialization*: this block is executed only the first time that some message for a (new) entity is received; useful to, e.g., assign initial values that are specific for single entities.
MessagePreprocessing*: this algorithm is executed every time a new message for an uncorrupted entity is received.
Main: specification of the actual behavior of an uncorrupted entity.
Procedures and Functions*: this block can be used to specify functions or procedures that can be used by other algorithms of this machine. Useful to, e.g., split up longer algorithms or reduce code repetitions.

Figure 3.1: Template for specifying protocols, see [6]. Blocks labeled with an asterisk (*) are optional. Our template differs to [6] by the addition of the block **Procedures and Functions**.

Exchanging Messages. Entities can send and receive messages using the `I/O` and network interfaces belonging to their respective role. The receiver of the message must always be specified, which is either the adversary in case of network interfaces of some other entity (with a role that has a connected `I/O` interface) in case of the `I/O` interface.

The iUC model specifies a convenient template for specifying protocols. This is illustrated in Figure 3.1.

3.1.2 Modeling Corruption

Depending on the used corruption model of a protocol, an entity might become explicitly corrupted by the adversary in a run of the system. In this case, the adversary gains full control of the corrupted entity: arriving messages at an entity are forwarded to the adversary, while the adversary can also tell an entity to send messages to other entities on behalf of the corrupted entity. The **AllowAdvMessage** algorithm can be used to restrict the adversary from sending messages on behalf of a corrupted entity to arbitrary entities.

However, the adversary cannot corrupt incorruptible protocols. An entity might consider itself implicitly corrupted, but in this case, the adversary does not gain control over the implicitly corrupted entity. This can be specified in the **DetermineCorrStatus** algorithm.

3.2 An iUC Model for Hashgraph

In this chapter we give a introduction to our iUC model, which is specified at the end of this chapter. We begin defining the Hashgraph protocol:

Definition 1 (The Hashgraph Protocol $\mathcal{P}^{\mathsf{H}}$).
For protocols $\mathcal{P}^{\mathsf{H}}_{\mathtt{client}}, \mathcal{P}^{\mathsf{H}}_{\mathtt{node}}, \mathcal{F}_{\mathtt{cert}}, \mathcal{F}_{\mathtt{ro}}, \mathcal{F}_{\mathtt{clock}}, \mathcal{F}_{\mathrm{init}}, \mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$ as in Figures 3.2 to 3.16 we define the Hashgraph protocol to be

$$\mathcal{P}^{\mathsf{H}} = (\mathtt{client}, \mathtt{init} \mid \mathtt{node}, \mathtt{cert}, \mathtt{ro}, \mathtt{clock}, \mathtt{judge}).$$

The protocols $\mathcal{P}^{\mathsf{H}}_{\mathtt{client}}$ and $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}}$ correspond to the clients and nodes as in Chapter 2. With $\mathcal{F}_{\mathrm{init}}$ we introduce an ideal initialization functionality that starts a hashgraph session. We stress that our model is able to run multiple hashgraph session in parallel. However, we consider a participant $pid \in \mathsf{nodes}$ to be dishonest if one of its node instances is corrupted. That is, the attacker explicitly corrupted a node instance of pid or a node instance considers itself implicitly corrupted due to the corruption of the signing key of its signer entity.

3.2.1 The Ideal Signature Functionality $\mathcal{F}_{\mathtt{cert}}$

The ideal protocol $\mathcal{F}_{\mathtt{cert}}$ is an incorruptible signature and verifier protocol that is used for two purposes: *(i)* Nodes can use the `signer` role of this protocol to sign arbitrary messages, and *(ii)* other participants can use the `verifier` role to check the validity of signatures. By default, $\mathcal{F}_{\mathtt{cert}}$ prevents forgeries of messages that have not been previously signed by the participant of the signing key. However, we still allow the network attacker to corrupt signing keys and give the attacker the ability to forge signatures. This will be explained in more detail later.

Entities in $\mathcal{F}_{\mathtt{cert}}$
Observe in the definition of $\mathcal{F}_{\mathtt{cert}}$ that one machine instance of $\mathcal{F}_{\mathtt{cert}}$ manages all parties and roles in a single session sid'. This session is a triple $(pid, sid, role)$ and thus represents that one machine instance of $\mathcal{F}_{\mathtt{cert}}$ manages the ideal signing and verifying functionalities of participant pid in one session sid and one role $role$. For a machine instance managing all entities with SID $(pid, sid, role)$, we call pid also the pidowner; this becomes important when signing messages.

Verifying signatures
Other entities of machines implementing $\mathcal{F}_{\mathtt{cert}}$ as subroutine ($M_{\mathtt{node}}$, $M_{\mathtt{client}}$, and $M_{\mathtt{judge}}$ in our case) can verify if a signature σ, supposedly signed by pid (in session sid with role $role$), of msg is indeed valid. This is done by sending the message $(\mathtt{Verify}, msg, \sigma)$ to $e = (\mathsf{pid}_{\mathsf{cur}}, (pid, sid, role), \mathcal{F}_{\mathtt{cert}} : \mathtt{verify})$, where $\mathsf{pid}_{\mathsf{cur}}$ is the party ID of the calling entity and the receiver is the machine instance of $\mathcal{F}_{\mathtt{cert}}$ that manages all entities with SID $(pid, sid, role)$. We denote the reply of entity e as

$$\mathtt{verifySig}_{\mathsf{pk}(pid)}(msg, \sigma).$$

One can observe such verification requests in the protocols $\mathcal{P}^{\mathsf{H}}_{\mathtt{client}}$, $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}}$, and $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$.

Signing messages
Similar to verifying signatures, participants (i.e., an entity $(pid, sid, role)$) can sign arbitrary messages msg at $\mathcal{F}_{\mathtt{cert}}$ by sending the message $(\mathtt{Sign}, msg)$ to $e = (\mathsf{pid}_{\mathsf{cur}}, (pid, sid, role), \mathcal{F}_{\mathtt{cert}} : \mathtt{signer})$ on the `I/O` interface. Analogously, we denote the reply of e as

$$\mathtt{sign}_{\mathsf{pk}(pid)}(msg).$$

Notice in our iUC model for Hashgraph that only nodes implement the $\mathtt{signer}$ role of $\mathcal{F}_{\mathtt{cert}}$ as subroutine. Therefore, only nodes are able to send messages to entities of role $\mathtt{signer}$. Moreover, a receiving entity $(pid, sid, \mathtt{signer})$ of $\mathcal{F}_{\mathtt{cert}}$ must be the pidowner itself; this is used to prevent nodes of other parties to sign messages for pid. But a node with party ID pid' could still send a message $(\mathtt{Sign}, msg)$ to entity $(pid, (pid, sid, role), \mathcal{F}_{\mathtt{cert}} : \mathtt{signer})$, where $pid \neq pid'$ and thus sign in the name of pid. One can observe in Figure 3.16 that honest nodes never sign messages of other nodes. However, to ensure the same for corrupted nodes, we restrict in the implementation of $\mathbf{AllowAdvMessage}$ in $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}}$ the adversary to send messages via an explicitly corrupted entity $(pid, sid, \mathcal{P}^{\mathsf{H}}_{\mathtt{node}} : \mathtt{node})$ to $(pid_{receiver}, sid_{receiver}, role_{receiver})$ if $pid \neq pid_{receiver}$. We will discuss more about this later in Chapter 5.

Description of $\mathcal{P}^{\mathsf{H}}_{\mathtt{client}} = (\mathtt{client})$:

Participating roles: $\{\mathtt{client}\}$
Corruption model: *Dynamic corruption without secure erasures*
Protocol parameters:
- $\eta \in \mathbb{N}$. $\{$*The security parameter, defines the bit-length of* $txId$ *and* $eventID$.
- $\mathsf{nodes} \subseteq \{0,1\}^*$. $\{$*The identities of the Hashgraph nodes.*

Description of $M_{\mathtt{client}}$:

Implemented role(s): $\{\mathtt{client}\}$
Subroutines: $\mathcal{F}_{\mathtt{cert}} : \mathtt{verifier}, \mathcal{F}^{\mathsf{H}}_{\mathtt{judge}} : \mathtt{judge}, \mathcal{F}_{\mathrm{init}} : \mathtt{init}$
Internal state:
- $\mathsf{states}\colon \mathsf{nodes} \to (\mathbb{N} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^*) \cup \{\emptyset\}$. $\{$*Mapping from nodes to states provided by these nodes; initially* $\emptyset$ *for all entries. A state is a list of transaction with entries of form* $(counter, txId, eventID, tx)$*; cf.* $\mathsf{msglist}$ *in* $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}}$.

CheckID$(pid, sid, role)$**:**
Accept all messages with the same SID and PID. $\{$*Accept a single party in one session.*

Corruption behavior:
DetermineCorrStatus$(pid, sid, role)$:
return $\mathsf{explicitCorr}[\mathsf{entity}_{\mathsf{cur}}]$[a].

Main:

```
recv (Submit, msg) from I/O:                        {Process a transaction submission. Any message is a valid transaction.
    send (Submit, msg) to NET;                      {The adversary is responsible for dispatching Submit requests.

recv Read from I/O:                                 {Process a read request.
    msglist ← ∅; i ← 1.
    while ∃e ∈ {i} × {0,1}^η × {0,1}^η × {0,1}* s.t. hasSupermajority(e) = true do
        msglist ← msglist ∪ {e}.
        i ← i + 1.
    output ← {(counter, msg) | (counter, _, _, msg) ∈ msglist}.      {Remove txId and eventID from output.
    reply (Read, output).

recv (StateUpdate, (pid_send, msglist), σ) from NET s.t. pid_send ∈ nodes:   {State update from a node – modeled as push operation.
    sigIsValid ← verifySig(pid_send, (pid_send, msglist), σ).
    if sigIsValid = true:                           {Drop state update if signature is invalid.
        isConsecutive ← true.
        for i = 1, ..., |msglist| do:
            if (i, _, _, _) ∉ msglist:              {If the reported msglist from pid is no consecutive sequence of
                isConsecutive ← false.               transaction, starting at index 1 and ending at index |msglist|,
                                                     decline update.
        if isConsecutive = true:
            if states[pid] ⊊ msglist:               {Messages with "older" state information are irgnored.
                states[pid] ← msglist.              {Record that pid_cur accepted an update from pid_send.

        send (Evidence, (pid_send, msglist), σ) to (pid_cur, sid_cur, F^H_judge : judge);

                       {Honest clients always report all state updates signed by pid_send to F^H_judge as evidence.
```

Procedures and Functions:

```
function hasSupermajority(entry) :                  {Check if entry(i, txId, eventID, tx) is in
    voteCount ← |{pid ∈ nodes | entry ∈ states[pid]}|.   at least 3/2|nodes| states of all nodes.
    if voteCount > 3/2|nodes|:
        return true.
    else:
        return false.

function verifySig(pid, msg, σ) :                   {Verify signature at F_cert.
    send (Verify, msg, σ) to (pid_cur, (pid, sid_cur, P^H_node : node), F_cert : verifier);
    wait for (VerResult, result).
    return result.
```

[a] $\mathsf{explicitCorr}$ is an internal framework specific variable in iUC that stores all explicitly corrupted entities (by adversary $\mathcal{A}$) that are managed by the current machine instance, see [6].

Figure 3.2: The model of a Hashgraph client $\mathcal{P}^{\mathsf{H}}_{\mathtt{client}}$ (Part 1).

Description of $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}} = (\mathtt{node})$:

Participating roles: $\{\mathtt{node}\}$
Corruption model: *Dynamic corruption without secure erasures*
Protocol parameters:
- $\eta \in \mathbb{N}$. {*The security parameter.*
- $\mathsf{nodes} \subseteq \{0,1\}^*$. {*The identities of the Hashgraph nodes.*
- $\mathsf{coinRound} \in \mathbb{N}$. {*After* coinRound *rounds in Hashgraph, there is a "coin round" during virtual voting.*

Description of $M_{\mathtt{node}}$:

Implemented role(s): $\{\mathtt{node}\}$
Subroutines: $\mathcal{F}_{\mathtt{cert}} : \mathtt{signer}, \mathcal{F}_{\mathtt{cert}} : \mathtt{verifier}, \mathcal{F}_{\mathtt{ro}} : \mathtt{randomOracle}, \mathcal{F}^{\mathsf{H}}_{\mathtt{judge}} : \mathtt{judge}, \mathcal{F}_{\mathtt{clock}} : \mathtt{clock}$
Internal state:
- $\mathsf{events} \subseteq (\mathsf{nodes} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*) \cup (\{\perp\}^7) = \emptyset$.
  {*This set represents the local hashgraph G of* $\mathsf{pid}_{\mathsf{cur}}$ *and is a subset of events with entries $(pid, eventID, selfParent, otherParent, transactions, ts, \sigma)$ or, in the case of the genesis event, $(\perp,\perp,\perp,\perp,\perp,\perp,\perp)$.*
- $\mathsf{msglist} \subseteq \mathbb{N} \cup \{\varepsilon\} \times \{0,1\}^\eta \times \{0,1\}^\eta \cup \{\varepsilon\} \times \{0,1\}^* = \emptyset$. {*A list of transactions: entries are of form $(counter, txId, eventID, tx)$.*
- $\mathsf{roundReceivedEvents} : \mathbb{N} \to (\mathsf{nodes} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*) \cup (\{\perp\}^7)$.
  {*Stores all events with a round received number in consecutive order, starting at index 0. Also used for sorting events; see function* `findOrder`.
- $\mathsf{lastEvent} \in \{0,1\}^\eta \cup \{\perp\} = \perp$. {*The last event created by* $\mathsf{pid}_{\mathsf{cur}}$.
- $\mathsf{roundCreated} : \{0,1\}^\eta \cup \{\perp\} \to \mathbb{N}$. {*The round created of an event (identified via $eventID$); initially 0 for all entries.*
- $\mathsf{roundReceived} : \{0,1\}^\eta \cup \{\perp\} \to \mathbb{N}$. {*The round received of an event (identified via $eventID$); initially 0 for all entries.*
- $\mathsf{tmpVotes} : \{0,1\}^\eta \to \{?, \mathtt{true}, \mathtt{false}\}$. {*Used to store votes during virtual voting; initially ? for all entries.*
- $\mathsf{eventConsensusNumber} : \{0,1\}^\eta \cup \{\perp\} \to \mathbb{N} \cup \{\varepsilon\}$. {*Stores the total order of events from* events*; intially ε for all entries.*
- $\mathsf{round} \in \mathbb{N}$. {*The current "time".*
- $\mathsf{isWitness} : \{0,1\}^\eta \cup \{\perp\} \to \{?, \mathtt{true}, \mathtt{false}\}$. {*Stores whether an event is a witness; intially ? for all entries.*
- $\mathsf{isFamous} : \{0,1\}^\eta \cup \{\perp\} \to \{?, \mathtt{true}, \mathtt{false}\}$. {*Stores whether an event is famous; intially ? for all entries.*

CheckID$(pid, sid, role)$**:**
Accept all entities with the same PID and SID. {*Accept a single party in one session.*

Corruption behavior:
DetermineCorrStatus$(pid, sid, role)$:
 if $\mathsf{entity}_{\mathsf{cur}} \in \mathsf{explicitCorr}$[a]**:** {*Check whether node itself is corrupted.*
  return `true`.
 return $\mathbf{corr}(pid, (pid, sid, \mathcal{P}^{\mathsf{H}}_{\mathtt{node}} : \mathtt{node}), \mathcal{F}_{\mathtt{cert}} : \mathtt{signer})$[b] {*Request corruption status at $\mathcal{F}_{\mathtt{cert}}$ and return its value.*

AllowAdvMessage$(pid, sid, role, pid_{receiver}, sid_{receiver}, role_{receiver}), m)$: [c]
 if $pid = pid_{receiver}$**:**
  return `true`.
 else:
  return `false`.

EntityInitialization:[d]
 $\mathsf{events.add}((\perp,\perp,\perp,\perp,\perp,\perp,\perp))$. {*Add the genesis event to the local hashgraph.*

Description of $M_{\mathtt{node}}$ continues in Figure 3.4.

[a] explicitCorr is an internal framework specific variable in iUC that stores all explicitly corrupted entities (by adversary $\mathcal{A}$) that are managed by the current machine instance, see [6].
[b] **corr** is a macro in iUC to querry if an entity is corrupted.
[c] We could have also omitted this algorithm entirely, as our implementation is the default behavior, specified in [6], if **AllowAdvMessage** is not stated explicitly.
[d] We could have also added this complete section to the "Initialization" block of iUC, since a machine manages exactly one entity.

Figure 3.3: The model of a Hashgraph node $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}}$ (Part 1).

Description of $M_{\texttt{node}}$ (cont.):

MessagePreprocessing:

recv msg **from** I/O or NET:
  if isInitialized $=$ true:
    send GetCurRound **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\texttt{clock}} : \texttt{clock})$; — *{Update internal clock.*
    wait for (GetCurRound, $round$).
    round $\leftarrow round$.
  else:
    abort. — *{Abort current machine activation if the client instance is not initialized. Upon executing the abort command, the environment gets activated by definition of the IITM model.*

Main:

recv (Submit, msg) **from** NET: — *{Process a transaction submission. Any message is a valid transaction.*
  $txId \leftarrow \texttt{hash}(msg)$. — *{Generate transaction ID.*
  msglist.add$(\varepsilon, txId, \varepsilon, msg)$. — *{Record submitted transaction.*

recv (RecvGossipEvents, $gossipedEvent$, $events$) **from** NET **s.t.** — *{Node receives a gossip event.*
    $events \subseteq (\mathsf{nodes} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^* \times \mathbb{N}\times) \cup (\{\bot\}^7)$ **:**
  $validity \leftarrow \texttt{verifyBasicGraphCorrectness}(events)$. — *{Ensure Basic Graph Correctness in Definition 6.*
  if $validity = \texttt{false}$:
    break. — *{If basic graph correctness is violated, break here.*
  if $\exists!(pid, eventID, \_, \_, \_, \_, \_) \in events$ **s.t.**
      $(pid \neq \mathsf{pid}_{\mathsf{cur}} \lor |events| = 1)$
      $\land\ eventID = gossipedEvent$**:** — *{Ensure complete other-ancestor construction for $eventID_{new}$ (see below) (cf. Definition 5, 1. (ii).)*
    if $\nexists(pid, eventID', selfPar, \_, \_, \_, \_) \in events$ **s.t.**
        $eventID' \neq eventID \land gossipedEvent = selfPar$**:** — *{Only "sync" with most recent event from pid, where pid is the creator of gossipedEvent.*
      for all $(\_, eventID'', \_, \_, txs, \_, \_) \in events \setminus (events \cap \mathsf{events})$ **do:**
        for all $msg \in txs$ **do:**
          msglist.add$((\varepsilon, \texttt{hash}(msg), eventID'', msg))$. — *{Add new messages to* msglist *from previous sync.*
      events.add$(events)$. — *{Merge new events into* $\mathsf{pid}_{\mathsf{cur}}$*'s hashgraph.*
      $txs_{new} \leftarrow \{msg \mid (\varepsilon, txId, \varepsilon, msg) \in \mathsf{msglist}\}$. — *{Add transactions for new event.*
      $eventID_{new} \leftarrow \texttt{hash}((\mathsf{lastEvent}, gossipedEvent, txs_{new}, \mathsf{round}))$. — *{Create eventID of the new event.*
      $\sigma_{new} \leftarrow \texttt{sign}((\mathsf{lastEvent}, gossipedEvent, txs_{new}, \mathsf{round}))$. — *{Sign new event.*
      events.add$((\mathsf{pid}_{\mathsf{cur}}, eventID_{new}, \mathsf{lastEvent}, gossipedEvent, txs_{new}, \mathsf{round}, \sigma_{new}))$.
        {Merge the new event ($eventID_{new}$) into local hashgraph.
      for all $(\varepsilon, txId, \varepsilon, msg) \in \mathsf{msglist}\}$ **do:**
        msglist.remove$((\varepsilon, txId, \varepsilon, msg))$.
        msglist.add$((\varepsilon, txId, eventID_{new}, msg))$. — *{Assign the newly created event ID to transactions in* msglist.
      lastEvent $\leftarrow eventID_{new}$.
      divideRounds(). — *{Determines rounds of events and whether events are witnesses.*
      decideFame(). — *{Determines whether events are "famous".*
      findOrder(). — *{Establishes total order over events and transactions.*
      send (EvidenceNode, events) **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}^{\mathsf{H}}_{\texttt{judge}} : \texttt{judge})$; — *{Report current state to* $\mathcal{F}^{\mathsf{H}}_{\texttt{judge}}$.

recv StateUpdate **from** NET: — *{Process a read request from* NET.
  $currentState \leftarrow \emptyset$.
  for all $(counter, txId, eventID, msg) \in \mathsf{msglist}$ **s.t.** $counter \neq \varepsilon$ **do:**
    $currentState$.add$((counter, txId, eventID, msg))$. — *{Extract output and drop eventID.*
  $\sigma \leftarrow \texttt{sign}((\mathsf{pid}_{\mathsf{cur}}, currentState))$. — *{Sign output.*
  reply (StateUpdate, $(\mathsf{pid}_{\mathsf{cur}}, currentState), \sigma$).

recv GossipEvents **from** NET: — *{$\mathcal{A}$ triggers when nodes gossip events.*
  $pid_r \xleftarrow{\$} \mathsf{nodes} \setminus \{\mathsf{pid}_{\mathsf{cur}}\}$. — *{Randomly chose a node to sync current events to.*
  reply (RecvGossipEvents, lastEvent, events).

recv GetCurRound **from** I/O or NET: — *{Current round can be requested externally.*
  reply (GetCurRound, round).

Description of $M_{\texttt{node}}$ continues in Figure 3.5.

Figure 3.4: The model of a Hashgraph node $\mathcal{P}^{\mathsf{H}}_{\texttt{node}}$ (Part 2).

Description of $M_{\texttt{node}}$ (cont.):

Procedures and Functions:

function `divideRounds`() **:** $\{$*Assign rounds to events and mark witnesses.*
for all $E \in$ events, *in topological order* [a] **do:**
$(\_, eventID, selfPar, otherPar, \_, \_, \_) \leftarrow$ parse E.
$r \leftarrow 1$.
if $eventID \neq \bot$**:** $\{$*Determine round created for events $E \neq$ genesis event.*
$r \leftarrow \max\{\mathsf{roundCreated}[selfPar], \mathsf{roundCreated}[otherPar]\}$.
$\mathcal{S} \leftarrow \{(\_, eventID', \_, \_, \_, \_, \_) \in \mathsf{events} \mid \mathsf{roundCreated}[eventID'] = r\}$.
$\mathcal{S} \leftarrow \{E' \in \mathcal{S} \mid \texttt{StronglySees}(E, E') = \texttt{true}\}$.
if $|\mathcal{S}| > \frac{2}{3}|\mathsf{nodes}|$**:**
$\mathsf{roundCreated}[eventID] \leftarrow r + 1$.
else:
$\mathsf{roundCreated}[eventID] \leftarrow r$.
if $(selfPar = \bot) \vee (\mathsf{roundCreated}[eventID] > \mathsf{roundCreated}[selfPar])$**:**
$\mathsf{isWitness}[eventID] \leftarrow \texttt{true}$. $\{$*If E is the genesis event, or it has the genesis event as self-parent, or it is the first event of pid within a new round, E is a witness.*

function `decideFame`() **:** $\{$*Decide which events are famous (cf. Figure 2.4).*
for all $E_r \in$ events, in order from earlier rounds to later **do:**
$(\_, eventID_r, \_, \_, \_, \_, \_) \leftarrow$ parse E_r $\{$*E_r is the candidate, i.e., the witness whom is voted for next.*
$\mathsf{tmpVotes} \leftarrow ?$ for all entries. $\{$*Initially, all votes for E_r are undecided.*
$\mathsf{isFamous}[eventID_r] \leftarrow ?$.
if $\mathsf{isWitness}[eventID_r] = \texttt{true}$**:** $\{$*Only witnesses take part in the election.*
$r \leftarrow \mathsf{roundCreated}[eventID_r]$.
for all $E_{r+1} \in \texttt{WitnessesOfRound}(r+1)$ **do:** $\{$*First round of the election.*
$(\_, eventID_{r+1}, \_, \_, \_, \_, \_) \leftarrow$ parse E_{r+1}.
if Sees[b]$(E_{r+1}, E_r) = \texttt{true}$**:** $\{$*All witnesses in the next round (w.r.t. E_r's round r) vote for E_r.*
$\mathsf{tmpVotes}[eventID_{r+1}] \leftarrow \texttt{true}$.
else:
$\mathsf{tmpVotes}[eventID_{r+1}] \leftarrow \texttt{false}$.
for all $i = 1, \ldots$ **s.t.** $(\mathcal{W}_{r+i+1} \leftarrow \texttt{WitnessesOfRound}(r+i+1)) \neq \emptyset$ **do:** $\{$*Further rounds of election.*
for all $E_{r+i+1} \in \mathcal{W}_{r+i+1}$ **do:**
$(\_, eventID_{r+i+1}, \_, \_, \_, \_, \sigma_{r+i+1}) \leftarrow$ parse E_{r+i+1}.
$\mathcal{W} \leftarrow \{E_{r+i} \in \texttt{WitnessesOfRound}(r+i) \mid \texttt{StronglySees}^c(E_{r+i+1}, E_{r+i}) = \texttt{true}\}$.
$y \leftarrow |\{(\_, eventID_{r+i}, \_, \_, \_, \_, \_) \in \mathcal{W} \mid \mathsf{tmpVotes}[eventID_{r+i}] = \texttt{true}\}|$.
$n \leftarrow |\{(\_, eventID_{r+i}, \_, \_, \_, \_, \_) \in \mathcal{W} \mid \mathsf{tmpVotes}[eventID_{r+i}] = \texttt{false}\}|$.
$\{$*All witnesses in the previous round (w.r.t. E_{r+i+1}'s round $r+i+1$) vote for E_r.*
if $y \geq \frac{|\mathcal{W}|}{2}|$**:**
$vote \leftarrow \texttt{true}$. $\{$*E_{r+i+1}'s vote, whether E_r is famous, is a majority voting among the witnesses it strongly sees.*
else:
$vote \leftarrow \texttt{false}$.
if $i \mod \mathsf{coinRound} > 0$**:** $\{$*This is a normal round.*
$\mathsf{tmpVotes}[eventID_{r+i+1}] \leftarrow vote$. $\{$*E_{r+i+1} votes in any case according to majority decision.*
if $y > \frac{2}{3}|\mathsf{nodes}| \vee n > \frac{2}{3}|\mathsf{nodes}|$**:** $\{$*If a supermajority is achieved during voting (i.e., $\frac{2}{3}|\mathsf{nodes}|$), E_{r+i+1} votes whether E_r is famous.*
$\mathsf{isFamous}[eventID_r] \leftarrow vote$.
break out of i-loop.
else: $\{$*Begin coin round.*
if $y > \frac{2}{3}|\mathsf{nodes}| \vee n > \frac{2}{3}|\mathsf{nodes}|$**:**
$\mathsf{tmpVotes}[eventID_{r+i+1}] \leftarrow vote$. $\{$*If supermajority is reached, E_{r+i+1} votes.*
else:
if last bit of σ_{r+i+1} is 1**:** $\{$*Otherwise, E_{r+i+1} "flips a coin".*
$\mathsf{tmpVotes}[eventID_{r+i+1}] \leftarrow \texttt{true}$.
else:
$\mathsf{tmpVotes}[eventID_{r+i+1}] \leftarrow \texttt{false}$.

Description of $M_{\texttt{node}}$ continues in Figure 3.6.

[a] Go over events in *topological order*, i.e., events are always "visited" after their parents.
[b] Returns `true` or `false` whether the first event (first argument) "can see" the second event (second argument).
[c] The function `StronglySees` returns `true` or `false` whether the first event "can strongly see" the second event.

Figure 3.5: The model of a Hashgraph node $\mathcal{P}^{\mathsf{H}}_{\texttt{node}}$ (Part 3).

Description of $M_{\texttt{node}}$ (cont.):

Procedures and Functions:

function `findOrder()` **:** {*Establish total order of events and transactions (cf. Figure 2.5).*

$maxRoundCreated \leftarrow 1$.

for all $(\_, eventID, \_, \_, \_, \_, \_) \in$ events **do:**

$maxRoundCreated \leftarrow \max\{maxRoundCreated, \text{roundCreated}[eventID]\}$.

for all $(E_1 \leftarrow (\_, eventID_1, \_, \_, \_, \_, \_)) \in$ events **do:**

for all $r = 1, \ldots, maxRoundCreated$ **do:** {*Check and calculate the round received number r of E_1, if it exists.*

if $\texttt{isRoundReceived}(E_1, r)$**:**

roundReceived$[eventID_1] \leftarrow r$. {*Round received number for E_1 exists.*

$\mathcal{S} \leftarrow \emptyset$. {*Intuitively, $\mathcal{S}$ contains events that other nodes included with, or shortly after, when they learned of E_1.*

for all $(E_2 \leftarrow (\_, eventID_2, selfPar_2, \_, \_, \_, \_)) \in$ events **do:**

$E_2\text{-}selfPar \leftarrow (\_, eventID, \_, \_, \_, \_, \_) \in$ events **s.t.** $eventID = selfPar_2$.

if $\exists(E_3 \leftarrow (\_, eventID_3, \_, \_, \_, \_, \_)) \in$ events **s.t.**

$\texttt{IsSelfAncestorOf}(E_2, E_3) = \texttt{true}$

$\wedge$ roundCreated$[eventID_3] = r$

$\wedge$ isFamous$[eventID_3] = \texttt{true}$[a]

$\wedge\ \texttt{IsUniqueFamousWitness}(E_3, r) = \texttt{true}$

$\wedge\ \texttt{IsAncestorOf}(E_1, E_2) = \texttt{true}$

$\wedge\ \texttt{IsAncestorOf}(E_1, E_2\text{-}selfPar) = \texttt{false}$ **:**

$\mathcal{S}$.add(E_2).

eventConsensusNumber$[eventID_1] \leftarrow \texttt{GetMedianWrtTimestamp}(\mathcal{S})$.

$counter \leftarrow 0$.

roundReceivedEvents $\leftarrow$? for all entries. {*Function that maps natural numbers to events; used for sorting events.*

for all $(E \leftarrow (\_, eventID, \_, \_, \_, \_, \_)) \in$ events **do:**

if roundReceived$[eventID] > 0$**:** {*Check if E has a round received number.*

roundReceivedEvents$[counter] \leftarrow E$. {*Add all events with a round received number to* `isRoundReceived`.

$counter \leftarrow counter + 1$.

{*Sort all events with a round received number as in Figure 2.5.*

roundReceivedEvents $\leftarrow$ `SortByRoundReceived`(roundReceivedEvents).

roundReceivedEvents $\leftarrow$ `SortTiesByEventConsensusNumber`(roundReceivedEvents).

roundReceivedEvents $\leftarrow$ `SortTiesBySignatureLexicographically`(roundReceivedEvents).

{*Now, all events are sorted.*

for all $(entry \leftarrow (ctr, txId, eventID, msg)) \in$ msglist **s.t.** $ctr \neq \varepsilon$ **do:** {*Removing sorted messages is not necessary, but it simplifies the proof for α_4 (ii).*

msglist.`remove`$(entry)$.

msglist.`add`$((\varepsilon, txId, eventID, msg))$.

$entryCounter \leftarrow 1$.

for $i = 0, \ldots$ **s.t.** roundReceivedEvents$[i] \neq$? **do:** {*Iterate over all sorted events in ascending order.*

$(\_, \_, \_, \_, txs, \_, \_) \leftarrow$ roundReceivedEvents$[i]$.

$M \leftarrow \{(\varepsilon, txId, eventID, msg) \in \text{msglist} \mid msg \in txs\}$. {*Set of all messages of the i-th event.*

for all $(\varepsilon, txId, eventID, msg) \in M$, in ascending order w.r.t. $txId$ **do:**

msglist.`add`$((entryCounter, txId, eventID, msg))$. {*Add an ordered entry to the local message list.*

msglist.`remove`$((\varepsilon, txId, eventID, msg))$. {*Remove the corresponding unordered entry.*

$entryCounter \leftarrow entryCounter + 1$.

function $\texttt{isRoundReceived}(E_1, r)$ **:** {*Checks if r fulfills all conditions for being a round received number of E_1 except for being minimal.*

for all $(E_2 \leftarrow (\_, eventID_2, \_, \_, \_, \_, \_)) \in$ events **do:**

if roundCreated$[eventID_2] \leq r$

$\wedge$ isWitness$[eventID_2] = \texttt{true}$

$\wedge$ isFamous$[eventID_2] =$? **:**

return `false`. {*The fame for all witnesses with a round created in or before round r must be decided.*

for all $E_3 \in \texttt{WitnessesOfRound}(r)$ **do:**

$(\_, eventID_3, \_, \_, \_, \_, \_) \leftarrow E_3$.

if $\texttt{IsAncestorOf}(E_1, E_3) = \texttt{false}$

$\wedge\ \texttt{IsUniqueFamousWitness}(eventID_3, r)$ **:**

return `false`. {*E_1 must be an ancestor of all round r unique famous witnesses.*

return `true`. {*Return* `true` *if all checks succeed.*

Description of $M_{\texttt{node}}$ continues in Figure Figure 3.7.

[a] In the absence of forking, each famous witness is also a unique famous witness (cf. [2]).

Figure 3.6: The model of a Hashgraph node $\mathcal{P}^{\mathsf{H}}_{\texttt{node}}$ (Part 4).

Description of $M_{\mathtt{node}}$ (cont.):

Procedures and Functions:

function $\mathtt{verifyBasicGraphCorrectness}(events)$: — *Check basic graph correctness for new synced events, see Definition 5.*
  for all $E \in events$ **do:**
    if $((pid, eventID, selfPar, otherPar, txs, ts, \sigma) \leftarrow E)$ can not be parsed **s.t.**
      1. $pid \in \mathsf{nodes}, txs, \sigma \in \{0,1\}^*, ts \in \mathbb{N}$
      2. $eventID, selfPar, otherPar \in \{0,1\}^\eta$ **:**
      if $E \neq (\bot,\bot,\bot,\bot,\bot,\bot,\bot)$**:**
        return $\mathtt{false}$. — *Nodes have to report well-formed evidence (cf. Definition 6, Well-Formed Data (i)); α_1.*
  for all $(pid, eventID, selfPar, otherPar, txs, ts, \sigma) \in \mathsf{events} \setminus \{(\bot,\bot,\bot,\bot,\bot,\bot,\bot)\}$ **do:**
    if $\neg(\exists(pid, eventID', \_,\_,\_,\_,\_) \in events$ **s.t.**
      $selfPar = eventID')$**:**
      return $\mathtt{false}$. — *All events, except for the genesis event, have to suffice a complete self-ancestor construction (cf. Definition 5, 1.(i)); α_2.*
    if $\neg(\exists(pid', eventID', \_,\_,\_,\_,\_)) \in events$ **s.t.**
      $otherPar = eventID' \wedge pid' \neq pid)$**:**
      return $\mathtt{false}$. — *All events, except for the genesis event, must satisfy a complete other-ancestor construction (cf. Definition 5, 1.(ii)); α_2.*
    if $\mathtt{verifySig}(pid, (selfPar, otherPar, txs, ts), \sigma) = \mathtt{false}$**:**
      return $\mathtt{false}$. — *Nodes need to report events with valid signatures (cf. Definition 5, 2.); α_2.*
    if $\mathtt{hash}((selfPar, otherPar, txs, ts)) \neq eventID$**:**
      return $\mathtt{false}$. — *Nodes must report valid events (cf. Definition 5, 3.); α_2.*
  if $(\bot,\bot,\bot,\bot,\bot,\bot,\bot) \notin events$**:**
    return $\mathtt{false}$. — *Ensure legitimate root constructions (cf. Definition 5, 4.); α_2.*
  if $\exists(\_,\_,\bot,\bot,\_,\_,\_) \in events$ **s.t.**
    $(\_,\_,\bot,\bot,\_,\_,\_) \neq (\bot,\bot,\bot,\bot,\bot,\bot,\bot)$**:**
    return $\mathtt{false}$. — *Ensure legitimate root construction; α_2.*
  if $\exists(pid, eventID, \bot,\bot,\_,\_,\_), (pid, eventID', \bot,\bot,\_,\_,\_)$ **s.t.**
    $eventID \neq eventID'$**:**
    return $\mathtt{false}$. — *Ensure legitimate root construction; α_2.*
  return $\mathtt{true}$.

function $\mathtt{sign}(msg)$: — *Sign message at $\mathcal{F}_{\mathtt{cert}}$.*
  send $(\mathtt{Sign}, msg)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{P}^{\mathsf{H}}_{\mathtt{node}} : \mathtt{node}), \mathcal{F}_{\mathtt{cert}} : \mathtt{signer})$;
  wait for $(\mathtt{Signature}, \sigma)$.
  return σ.

function $\mathtt{verifySig}(pid, msg, \sigma)$: — *Verify signature at $\mathcal{F}_{\mathtt{cert}}$.*
  send $(\mathtt{Verify}, msg, \sigma)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (pid, \mathsf{sid}_{\mathsf{cur}}, \mathcal{P}^{\mathsf{H}}_{\mathtt{node}} : \mathtt{node}), \mathcal{F}_{\mathtt{cert}} : \mathtt{verifier})$;
  wait for $(\mathtt{VerResult}, result)$.
  return $result$.

function $\mathtt{hash}(msg)$: — *Generate "hash" at $\mathcal{F}_{\mathtt{ro}}$.*
  send $(\mathtt{Hash}, msg)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathtt{ro}} : \mathtt{randomOracle})$;
  wait for $(\mathtt{Hash}, h)$.
  return h.

Figure 3.7: The model of a Hashgraph node $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}}$ (Part 5).

Description of the protocol $\mathcal{F}_{\texttt{cert}} = (\texttt{signer}, \texttt{verifier})$:

Participating roles: $\{\texttt{signer}, \texttt{verifier}\}$
Corruption model: *incorruptible*
Protocol parameters:
- $\mathsf{p} \in \mathbb{Z}[x]$. *{Polynomial that bounds the runtime of the algorithms provided by the adversary.*
- $\eta \in \mathbb{N}$. *{The security parameter.*
- sig. *{Signing algorithm, outputs a signature σ on input (msg, sk).*
- ver. *{Signature verifying algorithm, outputs verification result on input $(msg, \sigma, \mathsf{pk})$.*
- gen. *{Key generation algorithm, outputs $(\mathsf{pk}, \mathsf{sk})$ on input 1^η.*

Description of $M_{\texttt{signer},\texttt{verifier}}$:

Implemented role(s): $\{\texttt{signer}, \texttt{verifier}\}$
Internal state:
- $(\mathsf{pk}, \mathsf{sk}) \in (\{0,1\}^* \cup \{\bot\})^2 = (\bot, \bot)$. *{Key pair.*
- $\mathsf{pidowner} \in \{0,1\}^* \cup \{\bot\} = \bot$. *{Party ID of the key owner.*
- $\mathsf{msglist} \subseteq \{0,1\}^* = \emptyset$. *{Set of recorded messages.*
- $\mathsf{corrupted} \in \{\texttt{true}, \texttt{false}\} = \texttt{false}$. *{Is signature key corrupted?*

CheckID$(pid, sid, role)$**:**
Check if sid can be parsed as $(pid', sid', role')$:
If this check fails, output `reject`.
Otherwise, accept all entities with the same SID. *{A single instance manages all parties and roles in a single session. The session ID models one signature key pair belonging to party pid in a session sid.*

Corruption behavior:
- **DetermineCorrStatus**$(pid, sid, role)$:
  return corrupted.

Initialization:
$(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{Gen}(1^\eta)$. *{Generate public/secret key pair.*
$(pid, sid, role) \leftarrow$ parse $\mathsf{sid_{cur}}$.
$\mathsf{pidowner} \leftarrow pid$. *{Assign key owner.*

Main:

recv $(\texttt{Sign}, msg)$ **from** I/O **to** $(\mathsf{pidowner}, \_, \texttt{signer})$**:** *{Sign message; only pidowner is permited to sign with its key.*
  $\sigma \leftarrow \mathsf{sig}^{(\mathsf{p})}(msg, \mathsf{sk})$.
  $\mathsf{msglist.add}(msg)$.
  reply $(\texttt{Signature}, \sigma)$. *{Record msg for verification and return signature.*

recv $(\texttt{Verify}, msg, \sigma)$ **from** I/O **to** $(\_, \_, \texttt{verifier})$**:**
  $b \leftarrow \mathsf{ver}^{(\mathsf{p})}(msg, \sigma, \mathsf{pk})$. *{Verify signature.*
  if $b = \texttt{true} \wedge msg \notin \mathsf{msglist} \wedge \mathsf{corrupted} = \texttt{false}$**:**
    reply $(\texttt{VerResult}, \texttt{false})$. *{Prevent forgery.*
  else:
    reply $(\texttt{VerResult}, b)$. *{Return verification result.*

recv `corruptSigKey` **from** NET**:** *{Allow network attacker to corrupt signature keys.*
  $\mathsf{corrupted} \leftarrow \texttt{true}$.
  reply $(\texttt{corruptSigKey}, \texttt{ok})$.

Figure 3.8: The ideal signature functionality $\mathcal{F}_{\texttt{cert}}$ (cf. [11]).

Description of the protocol $\mathcal{F}_{\texttt{ro}} = (\texttt{randomOracle})$:

Participating roles: $\{\texttt{randomOracle}\}$
Corruption model: *incorruptible*
Protocol parameters:
- $\eta \in \mathbb{N}$. {*The security parameter and length of the hash.*

Description of $M_{\texttt{randomOracle}}$:

Implemented role(s): $\{\texttt{randomOracle}\}$
Internal state:
- hashHistory $\subseteq \{0,1\}^* \times \{0,1\}^\eta = \emptyset$. {*The set of recorded value/hash pairs; initially* $\emptyset$.

CheckID$(pid, sid, role)$**:**
Accept all messages with the same SID. {*One random oracle is responsible for one session of hashgraph.*

Main:

recv $(\texttt{Hash}, x)$ **from** I/O or NET**:** {*Requesting* $\mathcal{F}_{\texttt{ro}}$ *for "hashes".*
  if $\exists h \in \{0,1\}^\eta$ **s.t.** $(x, h) \in$ hashHistory**:** {*Extract existing value from* hashHistory.
    reply h.
  else:
    $h \xleftarrow{\$} \{0,1\}^\eta$ {*Generate "hash value" uniformly at random.*
    hashHistory $\leftarrow$ hashHistory.add$((x, h))$ {*Store generated key-value pair in* hashHistory.
    reply $(\texttt{Hash}, h)$.

Figure 3.9: The random oracle $\mathcal{F}_{\texttt{ro}}$ (cf. [7]).

Description of the ideal clock $\mathcal{F}_{\texttt{clock}} = (\texttt{clock})$:

Participating roles: $\{\texttt{clock}\}$
Corruption model: *incorruptible*

Description of $M_{\texttt{clock}}$:

Implemented role(s): $\{\texttt{clock}\}$
Internal state:
- $\tau \in \mathbb{N} = 0$. {*Current time in the* $\mathcal{F}_{\texttt{clock}}$*; initially* 0.

CheckID$(pid, sid, role)$**:**
Accept all messages for the same SID.

Main:

recv UpdateRound **from** I/O or NET**:** {*Triggering a clock update increases the time.*
  $\tau \leftarrow \tau + 1$.

recv GetCurRound **from** I/O or NET**:** {*Handling reads from the clock.*
  reply $(\texttt{GetCurRound}, \tau)$.

Figure 3.10: The ideal clock functionality $\mathcal{F}_{\texttt{clock}}$.

Description of the protocol $\mathcal{F}^{\mathsf{H}}_{\text{init}} = (\mathtt{init})$:

Participating roles: $\{\mathtt{init}\}$
Corruption model: *incorruptible*
Protocol parameters:
- $\mathsf{nodes} \subseteq \{0,1\}^*$. $\{$*The identities of the Hashgraph nodes in all sessions of hasgraph instances.*

Description of $M_{\mathtt{init}}$:

Implemented role(s): $\{\mathtt{init}\}$
Subroutines: $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}} : \mathtt{node}$
CheckID$(pid, sid, role)$**:**
Accept all entities with the same SID. $\{$*By this construction, we allow multiple Hashgraph instances to run parallel in one run of the system.*

Initialization:
for all $pid \in \mathsf{nodes}$ **do:**
init$(pid, \mathsf{sid}_{\mathsf{cur}}, \mathcal{P}^{\mathsf{H}}_{\mathtt{node}} : \mathtt{node})$[a]. $\{$*Initialize all nodes in the current Hashgraph instance; such instance is uniquely identified by the session ID of* $\mathsf{sid}_{\mathsf{cur}}$*, i.e., all nodes have the same SID* $\mathsf{sid}_{\mathsf{cur}}$*. Notice that* $\mathcal{P}^{\mathsf{H}}_{\mathtt{client}}$*,* $\mathcal{F}_{\mathtt{cert}}$*,* $\mathcal{F}_{\mathtt{ro}}$*, and* $\mathcal{F}_{\mathtt{clock}}$ *do not have to be initialized.*

Main: $\{$*Do nothing after initialization of the Hashgraph instance.*

[a] **init** is a macro in iUC to initialize an entity.

Figure 3.11: The ideal initialization functionality $\mathcal{F}^{\mathsf{H}}_{\text{init}}$.

Description of $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}} = (\mathtt{judge})$:

Participating roles: $\{\mathtt{judge}\}$
Corruption model: *incorruptible*
Protocol parameters:
- $\eta \in \mathbb{N}$. $\{$*The security parameter.*
- $\mathsf{nodes} \subseteq \{0,1\}^*$. $\{$*The identities of the Hashgraph nodes.*
- $\mathsf{coinRound} \in \mathbb{N}$. $\{$*After* coinRound *rounds in Hashgraph, there is a "coin round" during virtual voting.*

Description of $M_{\mathtt{judge}}$:

Implemented role(s): $\{\mathtt{judge}\}$
Subroutines: $\mathcal{F}_{\mathrm{sig}} : \mathtt{verifier}, \mathcal{F}_{\mathrm{ro}} : \mathtt{randomOracle}$
Internal state:
- $\mathsf{W} \subseteq \mathbb{N} \times \mathsf{nodes} \times ((\mathsf{nodes} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*) \cup (\{\bot\}^7)) = \emptyset$.
  $\{$*The collected evidences are of form (ctr, pid, G), where G is a hashgraph with entries as in $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}}$ (cf.* events *in Figure 3.3). Initially $\emptyset$, since there is no evidence at the beginning.*
- $\mathsf{states}\colon \mathsf{nodes} \to \mathbb{N} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^*$.
  $\{$*Mapping from nodes to states provided by these nodes; initially $\emptyset$ for all entries. A state is a list of transaction with entries of form $(counter, txId, , eventID, tx)$; cf.* msglist *in* $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}}$.
- $\mathsf{evidenceCounter} \in \mathbb{N} = 0$. $\{$*Counter for internally sorting evidence data; initially 0.*
- $\mathsf{msglist_{max}} \subseteq \mathbb{N} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^* = \emptyset$. $\{$*A list of transactions: entries are of form $(counter, txId, eventID, tx)$; initially $\emptyset$.*
- verdicts[a] $\subseteq \{0,1\}^* = \emptyset$. $\{$*Set of recorded verdicts; initially $\emptyset$.*
- $\mathsf{forkingNodes} \subseteq \mathsf{nodes} = \emptyset$. $\{$*Set of all node participants that created a fork.*
- $\mathsf{consistencyVerdicts} \subseteq \mathsf{nodes} = \emptyset$. $\{$*Set of all node participants that violated self or node-consistency (cf. γ_1, γ_2 in Definition 11).*
- $\mathsf{events} \subseteq (\mathsf{nodes} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*) \cup (\{\bot\}^7) = \emptyset$.
  $\{$*Set of events of form $(pid, eventID, selfParent, otherParent, transactions, ts, \sigma)$ – solely used for calculating purposes in* `divideRounds`, `decideFame`, *and* `findOrder`.
- $\mathsf{msglist} \subseteq \mathbb{N} \cup \{\varepsilon\} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^* = \emptyset$. $\{$*A list of transactions: entries are of form $(counter, txId, eventID, tx)$ – only used for internal computations.*
- $\mathsf{roundReceivedEvents} : \mathbb{N} \to (\mathsf{nodes} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*) \cup (\{\bot\}^7)$.
  $\{$*Stores all events with a round received number in consecutive order, starting at index 0. Also used for sorting events; see function* `findOrder`.
- $\mathsf{roundCreated} : \{0,1\}^\eta \cup \{\bot\} \to \mathbb{N}$. $\{$*The round created of an event (identified via $eventID$); initially 0 for all entries.*
- $\mathsf{roundReceived} : \{0,1\}^\eta \cup \{\bot\} \to \mathbb{N}$. $\{$*The round received of an event (identified via $eventID$); initially 0 for all entries.*
- $\mathsf{tmpVotes} : \{0,1\}^\eta \to \{?, \mathtt{true}, \mathtt{false}\}$. $\{$*Used to store votes during virtual voting; intially ? for all entries.*
- $\mathsf{eventConsensusNumber} : \{0,1\}^\eta \cup \{\bot\} \to \mathbb{N} \cup \{\varepsilon\}$. $\{$*Stores the total order of events from* events*; intially ε for all entries.*
- $\mathsf{isWitness} : \{0,1\}^\eta \cup \{\bot\} \to \{?, \mathtt{true}, \mathtt{false}\}$. $\{$*Stores whether an event is a witness; intially ? for all entries.*
- $\mathsf{isFamous} : \{0,1\}^\eta \cup \{\bot\} \to \{?, \mathtt{true}, \mathtt{false}\}$. $\{$*Stores whether an event is famous; intially ? for all entries.*

CheckID$(pid, sid, role)$**:**
Accept all messages with the same SID.

MessagePreprocessing:

Description of $M_{\mathtt{judge}}$ continues in Figure 3.13.

[a] Multiple verdicts of $\mathcal{F}_{\mathtt{judge}}$ should be interpreted as "and conected", e.g., for $\mathsf{verdicts} = \{\mathsf{dis}(pid_1), \mathsf{dis}(pid_2)\}$, $\mathcal{F}_{\mathtt{judge}}$ states (implicitly) the verdict $\mathsf{dis}(pid_1) \wedge \mathsf{dis}(pid_2)$.

Figure 3.12: The judging functionality $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$ for the Hashgraph model (Part 1).

Description of $M_{\mathtt{judge}}$ (cont.):

MessagePreprocessing:

recv msg **from** I/O**:**
  if $\nexists pid \in \mathsf{nodes}$ **s.t.** $\mathsf{dis}(pid) \notin \mathsf{verdicts}$**:**
    abort. {*If there exists no honest party, abord.*

Main:

recv GetVerdicts **from** I/O or NET**:** {*Current verdict can be requested externally.*
  reply (GetVerdicts, verdicts).

recv (EvidenceNode, $events$) **from** I/O **s.t.** $\mathsf{pid_{call}}$[a] $\in \mathsf{nodes}$**:** {*Process evidence from nodes.*
  for all $E \in events$ **do:**
    if $((pid, eventID, selfPar, otherPar, txs, ts, \sigma) \leftarrow E)$ can not be parsed **s.t.**
      1. $pid \in \mathsf{nodes}$, $txs, \sigma \in \{0,1\}^*$, $ts \in \mathbb{N}$
      2. $eventID, selfPar, otherPar \in \{0,1\}^\eta$ **:**
      if $E \neq (\bot,\bot,\bot,\bot,\bot,\bot,\bot)$**:**
        verdicts.add(dis($\mathsf{pid_{call}}$)). {*Nodes have to report well-formed evidence (cf. Definition 6, Well-Formed Data (i)); α_1.*
        break.
  W.add(evidenceCounter, $\mathsf{pid_{call}}$, $events$). {*Record new (witness) data.*
  evidenceCounter $\leftarrow$ evidenceCounter $+ 1$.

  for all $(pid, eventID, selfPar, otherPar, txs, ts, \sigma) \in \mathsf{events} \setminus \{(\bot,\bot,\bot,\bot,\bot,\bot,\bot)\}$ **do:**
    if $\neg(\exists(pid, eventID', \_,\_,\_,\_,\_) \in events$ **s.t.**
      $selfPar = eventID')$**:**
      verdicts.add(dis($\mathsf{pid_{call}}$)). {*All events, except for the genesis event, have to suffice a complete self-ancestor construction (cf. Definition 5, 1.(i)); α_2.*

    if $\neg(\exists(pid', eventID', \_,\_,\_,\_,\_)) \in events$ **s.t.**
      $otherPar = eventID' \land pid' \neq pid)$**:**
      verdicts.add(dis($\mathsf{pid_{call}}$)). {*All events, except for the genesis event, must satisfy a complete other-ancestor construction (cf. Definition 5, 1.(ii)); α_2.*

    if $\mathtt{verifySig}(pid, (selfPar, otherPar, txs, ts), \sigma) = \mathtt{false}$**:**
      verdicts.add(dis($\mathsf{pid_{call}}$)). {*Nodes need to report events with valid signatures (cf. Definition 5, 2.); α_2.*

    if $\mathtt{hash}((selfPar, otherPar, txs, ts)) \neq eventID$**:**
      verdicts.add(dis($\mathsf{pid_{call}}$)). {*Nodes must report valid events (cf. Definition 5, 3.); α_2.*

  if $(\bot,\bot,\bot,\bot,\bot,\bot,\bot) \notin events$**:**
    verdicts.add(dis($\mathsf{pid_{call}}$)). {*Ensure legitimate root constructions (cf. Definition 5, 4.); α_2.*
  if $\exists(\_,\_,\bot,\bot,\_,\_,\_) \in events$ **s.t.**
    $(\_,\_,\bot,\bot,\_,\_,\_) \neq (\bot,\bot,\bot,\bot,\bot,\bot,\bot)$**:**
    verdicts.add(dis($\mathsf{pid_{call}}$)). {*Ensure legitimate root construction; α_2.*
  if $\exists(pid, eventID, \bot,\bot,\_,\_,\_), (pid, eventID', \bot,\bot,\_,\_,\_)$ **s.t.**
    $eventID \neq eventID'$**:**
    verdicts.add(dis($\mathsf{pid_{call}}$)). {*Ensure legitimate root construction; α_2.*

  $\mathbb{E}_{\mathsf{W}} \leftarrow \bigcup_{(i,pid,G)\in \mathsf{W}} G$. {*Ensure fork-freeness for all participants $pid \in$ nodes (cf. Definition 8); β_1.*
  for $(pid, eventID_1, selfPar_1, otherPar_1, txs_1, ts_1, \sigma_1) \in \mathbb{E}_{\mathsf{W}}$ **do:**
    for $(pid, eventID_2, selfPar_2, otherPar_2, txs_2, ts_2, \sigma_2) \in \mathbb{E}_{\mathsf{W}}$ **do:**
      if $eventID_1 \neq eventID_2 \land selfPar_1 = selfPar_2$**:**
        if $\mathtt{verifySig}(pid, (selfPar_1, otherPar_1, txs_1, ts_1), \sigma_1) = \mathtt{true}$
          $\land\ \mathtt{verifySig}(pid, (selfPar_2, otherPar_2, txs_2, ts_2), \sigma_2) = \mathtt{true}$ **:**
          forkingNodes.add(pid).
          verdicts.add(dis(pid)). {*Ensure fork-freeness for all participants $pid \in$ nodes (cf. Definition 8); β_1.*

  for all $ctr = 0, \ldots,$ evidenceCounter $- 1$ **do:**
    if $(ctr, \mathsf{pid_{call}}, G') \in \mathsf{W}$**:**
      if $G' \not\subset events$**:**
        verdicts.add(dis($\mathsf{pid_{call}}$)). {*Each submitted graph G by $\mathsf{pid_{call}}$ has to be a proper superset of each prviously submitted graph G' by $\mathsf{pid_{call}}$ (cf. Definition 6, valid evidence); α_3.*

Description of $M_{\mathtt{judge}}$ continues in Figure 3.14.

[a] By definition of **AllowAdvMessage** in $M_{\mathtt{node}}$, it always applies $\mathsf{pid_{call}} = \mathsf{pid_{cur}}$

Figure 3.13: The judging functionality $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$ for the Hashgraph model (Part 2).

Description of $M_{\texttt{judge}}$ (cont.):

Main:

recv $(\texttt{EvidenceClient}, (pid_{send}, msglist), \sigma)$ **from** $\texttt{I/O}$ **s.t.** $pid_{send} \in$ nodes: {*Process evidence from clients.*
  $sigIsValid \leftarrow \texttt{verifySig}(pid_{send}, (pid_{send}, msglist), \sigma)$.
  if $sigIsValid = \texttt{false}$:
    break. {*Ensure that only valid evidence is processed.*
  if $msglist \not\subseteq \mathbb{N} \times \{0,1\}^{\eta} \times \{0,1\}^{\eta} \times \{0,1\}^{*}$:
    verdicts.add(dis(pid_{send}))
    break. {*The message list of clients must be well-formed (cf. Definition 6, Well-Formed Data (ii)); α_1.*
  for all $(ctr, txId, eventID, msg) \in msglist$ **do:**
    if $txId \neq \texttt{hash}(msg)$:
      verdicts.add(dis(pid_{send})).
      break. {*Nodes are obligated to compute valid hashes for all messages they include in their $msglist$ (cf. Definition 6, Valid Msglist (i)); α_4.*
  for $i = 1, \ldots, |msglist|$ **do:**
    if $(i, \_, \_, \_) \notin msglist$:
      verdicts.add(dis(pid_{send})).
      break. {*If the reported $msglist$ from pid is no consecutive sequence of transaction, starting at index 1 and ending at index $|msglist|$, decline update (cf. Definition 6, Valid Msglist (ii)); α_4.*
  if $pid_{send} \notin$ consistencyVerdicts $\wedge$ |forkingNodes| $< \frac{1}{3}$:
    if states[pid_{send}] $\subsetneq msglist$: {*Messages with "older" state information are irgnored.*
      states[pid_{send}] $\leftarrow msglist$. {*Record update for pid_{send}.*
      if $\text{msglist}_{\text{max}} \subsetneq$ states[pid_{send}]:
        $\text{msglist}_{\text{max}} \leftarrow$ states[pid_{send}]. {*Record that $msglist$ is now the longest known message list.*
    if $msglist \not\subseteq$ states[pid_{send}] $\wedge$ states[pid_{send}] $\not\subseteq msglist$:
      verdicts.add(dis(pid_{send})).
      consistencyVerdicts.add(pid_{send}). {*Node pid_{send} violated self-consistency (cf. Definition 11); γ_1.*
      $\text{msglist}_{\text{max}} \leftarrow \bigcup_{pid \in \text{nodes} \setminus \text{consistencyVerdicts}}$ states[pid]. {*Recalculate longest state of nodes that have not yet been blamed for consistency violations.*
    if states[pid_{send}] $\not\subseteq \text{msglist}_{\text{max}}$: {*Only true if $msglist_2$ and $\text{msglist}_{\text{max}}$ do not share a prefix.*
      $\texttt{generateReport}()$. {*Some participant(s) must have misbehaved w.r.t. node-consistency. Let us find the culprit in* `generateReport`.

Procedures and Functions:

function $\texttt{generateReport}()$: {*Recalculate the state of all nodes with their latest submitted hashgraphs.*
  for all $(ctr, pid, G) \in$ W **s.t.** dis(pid) $\notin$ verdicts **do:**
    if $\nexists (ctr', pid, G')$ **s.t.** $ctr' > ctr$: {*Calculate pid's state on the last hashgraph (set of events) which pid reported (cf. Definition 6, α_3).*
      events $\leftarrow G$; msglist $\leftarrow \emptyset$.
      roundCreated $\leftarrow 0$ for all entries.
      roundReceived $\leftarrow 0$ for all entries.
      isWitness $\leftarrow ?$ for all entries. {*Reset all necessary internal variables; note, not all variables, relevant to the consensus algorithm, need to be reset.*
      for all $(\_, eventID, \_, \_, txs, \_, \_) \in$ events **do:**
        for all $msg \in txs$ **do:**
          msglist.add(ε, $\texttt{hash}(msg)$, $eventID$, msg). {*Build message list of transactions from latest submitted hashgraph.*
      $\texttt{divideRounds}()$.
      $\texttt{decideFame}()$.
      $\texttt{findOrder}()$. {*Calculate total order of entries in* msglist*; the calculation is entirely identical to the calculations nodes perform internally.*
      if states[pid] $\not\subseteq$ msglist:
        verdicts.add(dis(pid)).
        consistencyVerdicts.add(pid_{send}). {*The latest message list of pid is not derivable from pid's latest reported internal state G – thus pid is responsible for the node-consistency violation (cf. Definition 11); γ_2.*
  $\text{msglist}_{\text{max}} \leftarrow \bigcup_{pid \in \text{nodes} \setminus \text{consistencyVerdicts}}$ states[pid]. {*Recalculate longest state of nodes that have not yet been blamed for consistency violations.*

function $\texttt{verifySig}(pid, msg, \sigma)$: {*Verify signature at $\mathcal{F}_{\texttt{cert}}$.*
  send $(\texttt{Verify}, msg, \sigma)$ **to** $(\text{pid}_{\text{cur}}, (pid, \text{sid}_{\text{cur}}, \mathcal{P}^{\mathsf{H}}_{\texttt{node}} : \texttt{node}), \mathcal{F}_{\texttt{cert}} : \texttt{verifier})$;
  wait for $(\texttt{VerResult}, result)$.
  return $result$.

function $\texttt{hash}(msg)$: {*Generate "hash" at $\mathcal{F}_{\texttt{ro}}$.*
  send $(\texttt{Hash}, msg)$ **to** $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\texttt{ro}} : \texttt{randomOracle})$;
  wait for $(\texttt{Hash}, h)$.
  return h.

Description of $M_{\texttt{judge}}$ continues in Figure 3.15.

Figure 3.14: The judging functionality $\mathcal{F}^{\mathsf{H}}_{\texttt{judge}}$ for the Hashgraph model (Part 3).

Description of $M_{\mathtt{judge}}$ (cont.):

Procedures and Functions:

```
function divideRounds() :                                    {Assign rounds to events and mark witnesses.
  for all E ∈ events, in topological order^a do:
    (_, eventID, selfPar, otherPar, _, _, _) ← parse E.
    r ← 1.
    if eventID ≠ ⊥:
      r ← max{roundCreated[selfPar], roundCreated[otherPar]}.   {Determine round created for
                                                                  events E ≠ genesis event.
    S ← {(_, eventID', _, _, _, _, _) ∈ events | roundCreated[eventID'] = r}.
    S ← {E' ∈ S | StronglySees(E, E') = true}.
    if |S| > 2/3|nodes|:
      roundCreated[eventID] ← r + 1.
    else:
      roundCreated[eventID] ← r.
    if (selfPar = ⊥) ∨ (roundCreated[eventID] > roundCreated[selfPar]):
                                   {If E is the genesis event, or it has the genesis event as self-parent, or
      isWitness[eventID] ← true.    it is the first event of pid within a new round, E is a witness.

function decideFame() :                                      {Decide which events are famous (cf. Figure 2.4).
  for all E_r ∈ events, in order from earlier rounds to later do:
    (_, eventID_r, _, _, _, _, _) ← parse E_r          {E_r is the candidate, i.e., the witness whom is voted for next.
    tmpVotes ← ? for all entries.                           {Initially, all votes for E_r are undecided.
    isFamous[eventID_r] ←?.
    if isWitness[eventID_r] = true:                         {Only witnesses take part in the election.
      r ← roundCreated[eventID_r].
      for all E_{r+1} ∈ WitnessesOfRound(r + 1) do:                     {First round of the election.
        (_, eventID_{r+1}, _, _, _, _, _) ← parse E_{r+1}.
        if Sees^b(E_{r+1}, E_r) = true:                {All witnesses in the next round (w.r.t. E_r's
          tmpVotes[eventID_{r+1}] ← true.               round r) vote for E_r.
        else:
          tmpVotes[eventID_{r+1}] ← false.
      for all i = 1,... s.t. (W_{r+i+1} ← WitnessesOfRound(r + i + 1)) ≠ ∅ do:   {Further rounds of election.
        for all E_{r+i+1} ∈ W_{r+i+1} do:
          (_, eventID_{r+i+1}, _, _, _, _, σ_{r+i+1}) ← parse E_{r+i+1}.
          W ← {E_{r+i} ∈ WitnessesOfRound(r + i) | StronglySees^c(E_{r+i+1}, E_{r+i}) = true}.
          y ← |{(_, eventID_{r+i}, _, _, _, _, _) ∈ W | tmpVotes[eventID_{r+i}] = true}|.
          n ← |{(_, eventID_{r+i}, _, _, _, _, _) ∈ W | tmpVotes[eventID_{r+i}] = false}|.
                                              {All witnesses in the previous round (w.r.t. E_{r+i+1}'s round
          if y ≥ |W|/2|:                        r + i + 1) vote for E_r.
            vote ← true.
          else:                               {E_{r+i+1}'s vote, whether E_r is famous, is a majority
            vote ← false.                       voting among the witnesses it strongly sees.
          if i mod coinRound > 0:                                         {This is a normal round.
            tmpVotes[eventID_{r+i+1}] ← vote.   {E_{r+i+1} votes in any case according to majority decision.
            if y > 2/3|nodes| ∨ n > 2/3|nodes|:   {If a supermajority is achieved during voting (i.e.,
              isFamous[eventID_r] ← vote.           2/3|nodes|), E_{r+i+1} votes whether E_r is famous.
              break out of i-loop.
          else:                                                           {Begin coin round.
            if y > 2/3|nodes| ∨ n > 2/3|nodes|:
              tmpVotes[eventID_{r+i+1}] ← vote.          {If supermajority is reached, E_{r+i+1} votes.
            else:
              if last bit of σ_{r+i+1} is 1:             {Otherwise, E_{r+i+1} "flips a coin".
                tmpVotes[eventID_{r+i+1}] ← true.
              else:
                tmpVotes[eventID_{r+i+1}] ← false.
```

Description of $M_{\mathtt{judge}}$ continues in Figure 3.16.

[a] Go over events in *topological order*, i.e., events are always "visited" after their parents.
[b] Returns `true` or `false` whether the first event (first argument) "can see" the second event (second argument).
[c] The function `StronglySees` returns `true` or `false` whether the first event "can strongly see" the second event.

Figure 3.15: The judging functionality $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$ for the Hashgraph model (Part 4).

Description of $M_{\texttt{judge}}$ (cont.):

Procedures and Functions:

function `findOrder()` **:** {*Establish total order of events and transactions (cf. Figure 2.5).*
  $maxRoundCreated \leftarrow 1$.
  for all $(\_, eventID, \_, \_, \_, \_, \_) \in$ events **do:**
    $maxRoundCreated \leftarrow \max\{maxRoundCreated, \text{roundCreated}[eventID]\}$.
  for all $(E_1 \leftarrow (\_, eventID_1, \_, \_, \_, \_, \_)) \in$ events **do:**
    for all $r = 1, \ldots, maxRoundCreated$ **do:** {*Check and calculate the round received number r of E_1, if it exists.*
      if `isRoundReceived`(E_1, r)**:**
        roundReceived$[eventID_1] \leftarrow r$. {*Round received number for E_1 exists.*
        $\mathcal{S} \leftarrow \emptyset$. {*Intuitively, $\mathcal{S}$ contains events that other nodes included with, or shortly after, when they learned of E_1.*
        for all $(E_2 \leftarrow (\_, eventID_2, selfPar_2, \_, \_, \_, \_)) \in$ events **do:**
          E_2-$selfPar \leftarrow (\_, eventID, \_, \_, \_, \_, \_) \in$ events **s.t.** $eventID = selfPar_2$.
          if $\exists(E_3 \leftarrow (\_, eventID_3, \_, \_, \_, \_, \_)) \in$ events **s.t.**
              `IsSelfAncestorOf`$(E_2, E_3) =$ `true`
              $\wedge$ roundCreated$[eventID_3] = r$
              $\wedge$ isFamous$[eventID_3] =$ `true`[a]
              $\wedge$ `IsUniqueFamousWitness`$(E_3, r) =$ `true`
              $\wedge$ `IsAncestorOf`$(E_1, E_2) =$ `true`
              $\wedge$ `IsAncestorOf`$(E_1, E_2\text{-}selfPar) =$ `false` **:**
            $\mathcal{S}$.`add`(E_2).
        eventConsensusNumber$[eventID_1] \leftarrow$ `GetMedianWrtTimestamp`$(\mathcal{S})$.
  $counter \leftarrow 0$.
  roundReceivedEvents $\leftarrow ?$ for all entries. {*Function that maps natural numbers to events; used for sorting events.*
  for all $(E \leftarrow (\_, eventID, \_, \_, \_, \_, \_)) \in$ events **do:**
    if roundReceived$[eventID] > 0$**:** {*Check if E has a round received number.*
      roundReceivedEvents$[counter] \leftarrow E$. {*Add all events with a round received number to* `isRoundReceived`.
      $counter \leftarrow counter + 1$.
  {*Sort all events with a round received number as in Figure 2.5.*
  roundReceivedEvents $\leftarrow$ `SortByRoundReceived`(roundReceivedEvents).
  roundReceivedEvents $\leftarrow$ `SortTiesByEventConsensusNumber`(roundReceivedEvents).
  roundReceivedEvents $\leftarrow$ `SortTiesBySignatureLexicographically`(roundReceivedEvents).
  {*Now, all events are sorted.*
  for all $(entry \leftarrow (ctr, txId, eventID, msg)) \in$ msglist **s.t.** $ctr \neq \varepsilon$ **do:** {*Removing sorted messages is not necessary, but it simplifies the proof for α_4 (ii).*
    msglist.`remove`$(entry)$.
    msglist.`add`$((\varepsilon, txId, eventID, msg))$.
  $entryCounter \leftarrow 1$.
  for $i = 0, \ldots$ **s.t.** roundReceivedEvents$[i] \neq ?$ **do:** {*Iterate over all sorted events in ascending order.*
    $(\_, \_, \_, \_, txs, \_, \_) \leftarrow$ roundReceivedEvents$[i]$.
    $M \leftarrow \{(\varepsilon, txId, eventID, msg) \in \text{msglist} \mid msg \in txs\}$. {*Set of all messages of the i-th event.*
    for all $(\varepsilon, txId, eventID, msg) \in M$, in ascending order w.r.t. $txId$ **do:**
      msglist.`add`$((entryCounter, txId, eventID, msg))$. {*Add an ordered entry to the local message list.*
      msglist.`remove`$((\varepsilon, txId, eventID, msg))$. {*Remove the corresponding unordered entry.*
      $entryCounter \leftarrow entryCounter + 1$.

function `isRoundReceived`(E_1, r) **:** {*Checks if r fulfills all conditions for being a round received number of E_1 except for being minimal.*
  for all $(E_2 \leftarrow (\_, eventID_2, \_, \_, \_, \_, \_)) \in$ events **do:**
    if roundCreated$[eventID_2] \leq r$
        $\wedge$ isWitness$[eventID_2] =$ `true`
        $\wedge$ isFamous$[eventID_2] = ?$ **:**
      return `false`. {*The fame for all witnesses with a round created in or before round r must be decided.*
  for all $E_3 \in$ `WitnessesOfRound`(r) **do:**
    $(\_, eventID_3, \_, \_, \_, \_, \_) \leftarrow E_3$.
    if `IsAncestorOf`$(E_1, E_3) =$ `false`
        $\wedge$ `IsUniqueFamousWitness`$(eventID_3, r)$ **:**
      return `false`. {*E_1 must be an ancestor of all round r unique famous witnesses.*
  return `true`. {*Return* `true` *if all checks succeed.*

[a] In the absence of forking, each famous witness is also a unique famous witness (cf. [2]).

Figure 3.16: The judging functionality $\mathcal{F}^{\mathsf{H}}_{\texttt{judge}}$ for the Hashgraph model (Part 5).

4 Accountability

Accountability is a security concept that does not prevent (directly) certain security goals from being violated. However, it does require that all involved misbehaving participants can be uniquely identified. This is achieved by means of a *judging procedure* or *judge*, which is a algorithm that outputs verdicts for misbehaving participants. By this, participants are incentivized to follow protocol rules.

For practical applications, it is usually necessary to indentify at least one individual participants for a misbehavior. If for instance, only a group of participants can be held accountable, and it is known that not all participants in this group must have misbehaved, then each party will deny responsibility, rendering this “weaker” accountability notion unsuitable for practical purposes. Therefore, at least one individual participant must be held accountable. The authors in [10, 14] refer to this as *individually accountability*. All of our proofs in Chapter 5 of accountability w.r.t. a security goal will fulfill this notion.

4.1 Formal Definition of Accountability

In the following, we aim to formalize the concepts and objectives of accountability. For this work, we apply the formal definition of accountability in [10], which adapts and enhances the accountability framework from [14] to (i) permit the judge to render a verdict at any point during a run and to (ii) allow dynamic corruption (in addition, to only static corruption).

Let $\mathbb{Q}$ be a system of machine instances (as in the iUC model) that includes a judge machine instace J, and let Σ denote the set of all participants in $\mathbb{Q}$. In the iUC framework, participants $p \in \Sigma$ are usually uniquely identified by party IDs (PIDs), and p typically belongs to exactly one instance of a machine in a run of $\mathbb{Q}$.

During a run of the system $\mathbb{Q}$, we expect J to render verdicts if a participent misbehaved. Formally, a *verdict* of J is a boolean formula ψ built from atomic propositions of the form $\mathsf{dis}(p)$ for $p \in \Sigma$, which indicates (a allegedly) misbehavior of p, i.e., p behaved dishonestly by violating prescribed protocol rules. Consider, for instance, the verdict $\mathsf{dis}(p_1) \vee \mathsf{dis}(p_2)$ by J; this states the judge's conviction that at least one of the participants p_1 and p_2 misconducted. Analogously, if J renders a verdict $\mathsf{dis}(p_1) \vee \mathsf{dis}(p_2)$, she claims that both p_1 and p_2 misbehaved. We denote the set of all verdicts by $\mathcal{V}$. In a run of $\mathbb{Q}$, the judge J can state multiple verdicts $\psi_1, \ldots, \psi_k \in \mathcal{V}$, which is semantically equivalent to J claiming the verdict $\psi_1 \wedge \ldots \wedge \psi_k$.

Since a verdict is a boolean formula, it can be evaluated to true or false using propositional logic. Let ω be a run of $\mathbb{Q}$ with $\lfloor\omega\rfloor$ denoting some point in this run. For a participant $p \in \Sigma$, the proposition $\mathsf{dis}(p)$ is set to true iff p is corrupted at point $\lfloor\omega\rfloor$, and set to false otherwise[1]. Formally, we will write $\lfloor\omega\rfloor \models \psi$ iff the verdict ψ is true at $\lfloor\omega\rfloor$; otherwise, we write $\lfloor\omega\rfloor \not\models \psi$.

In order to formalize and precisely capture the intended level of accountability, [14] introduces the notion of security properties and accountability constraints: A *security property* (or property) α of a protocol $\mathbb{Q}$ is a subset of all runs of $\mathbb{Q}$. Intuitively, α contains all runs of Q in which some predefined (security) goal is not satisfied as a consequence of some misbehaving protocol participant(s). By means of a property α of $\mathbb{Q}$, we define an *accountability constraint* C of a protocol $\mathbb{Q}$ to be a tuple $(\alpha, \psi_1, \ldots, \psi_k)$, written $(\alpha \Rightarrow \psi_1 \mid \cdots \mid \psi_k)$, with one or more verdicts $\psi_1, \ldots, \psi \in \mathcal{V}$. Intuitively, C specifies a set of minimal verdicts $\psi_1, \ldots, \psi_k$, wherein at least one of these verdicts are assumed to be rendered by J if some security property is violated (i.e., the run of the protocol is in α); however, J is free to state stronger verdicts that logically imply at least one of the minimal verdicts of C (in the sense of propositional logic). We say, for a run ω of $\mathbb{Q}$, that J *ensures C in ω* if either $\omega \notin \alpha$ or at some point in the run of ω the judge renders a verdict ψ that implies one of $\psi_1, \ldots, \psi_k$.

Example 2. We illustrate the notion of accountability constraints in the following examples, in which we consider n hashgraph nodes pid_i managed by different parties that are supposed to be blamed by J for any misbehavior defined by α:

$$C_1 \coloneqq (\alpha \Rightarrow \mathsf{dis}(pid_1) \mid \cdots \mid \mathsf{dis}(pid_n)) \tag{4.1}$$

$$C_2 \coloneqq (\alpha \Rightarrow \mathsf{dis}(pid_1) \vee \cdots \vee \mathsf{dis}(pid_n)). \tag{4.2}$$

In this work, we are primarily interested in assuring consistency for hashgraph; thus, the property α will later contain all runs of hashgraph where consistency is violated. Constraint C_1^{H} requires that if α (e.g., consistency) is violated, then at least one node pid_i can be held accountable by J in this run, i.e., J ensures C_1^{H} in a run $\omega \in \alpha$ by stating $\mathsf{dis}(pid_i)$ or, more generally, $\mathsf{dis}(pid_i) \bigwedge_{j\in\{1,\ldots,n\},j\neq i} \mathsf{dis}(pid_j)$ if multiple parties violated α (we stress that J is not obliged to render the more general verdict if multiple parties misbehaved; we merely require that at least one party is rightfully blamed in a run). A verdict of the form $\mathsf{dis}(pid_i) \bigvee_{j\in\{1,\ldots,n\},j\neq i} \mathsf{dis}(pid_j)$ by J is not sufficient to ensure C_1^{H}, since no individual participant can be blamed; however, J clearly ensures the weaker constraint C_2^{H}.

As discussed in the before, in practice it is necessary to ensure individual accountability. Formally, an accountability constraint $(\alpha \Rightarrow \psi_1 \mid \cdots \mid \psi_k)$ is said to achieve *individual accountability* if for every $i \in \{1, \ldots, k\}$ there exists a party $p \in \Sigma$ such that ψ_i implies $\mathsf{dis}(p)$. Therefore, each minimal verdict ψ_i determines at least one misbehaving party. In the example above, C_1^{H} certainly provides individual accountability, but C_2^{H} does not.

A set Φ of accountability constraints for the protocol $\mathbb{Q}$ is called an *accountability property* of $\mathbb{Q}$. Grouping different accountability constraints allows more expressiveness in defining security goals, but for our case study of hashgraph we will consider only a single constraint. Formally, we

[1]This enhances the definition in [14] by allowing dynamic corruption, i.e., corruption at arbitrary points in a run. Note that once a party gets corrupted at some point in a run, we consider it as corrupted for the rest of the run.

will write $\Pr[\mathbb{Q}(1^\eta) \mapsto \neg(J : \Phi)]$ to denote the probability that the judge J does not ensure C, for some $C \in \Phi$, in a run of $\mathbb{Q}(1^\eta)$, in which the probability is taken over the random coins of the run and 1^η is the security parameter given to the machine instances. Moreover, we denote by $\Pr[\mathbb{Q}(1^\eta) \mapsto \{(J : \psi) \mid \lfloor\omega\rfloor \not\models \psi\}]$ the probability that J states a verdict ψ at some point $\lfloor\omega\rfloor$ in a run of $\mathbb{Q}(1^\eta)$ such that $\lfloor\omega\rfloor \not\models \psi$, i.e., J renders a false verdict. Intuitively, we would like to demand from J to ensure all accountability constraints in Φ and never render false verdicts, except for a negligible number of cases; this is captured in the subsequent definition:

Definition 2 (Accountability).
Let $\mathbb{Q}$ be a system of machine instances with participants Σ that includes a judge $J \in \Sigma$, and let Φ be an accountability property of $\mathbb{Q}$. We say that J ensures Φ-accountability for System $\mathbb{Q}$ *(or $\mathbb{Q}$* is Φ-accountable w.r.t. *J) iff*

(i) (fairness) $\Pr[\mathbb{Q}(1^\eta) \mapsto \{(J : \psi) \mid \lfloor\omega\rfloor \not\models \psi\}]$ *is negligible as a function in η*[2]*, and*

(ii) (completeness) $\Pr[\mathbb{Q}(1^\eta) \mapsto \neg(J : \Phi)]$ *is negligible as a function in η.*

[2] A function $f : \mathbb{N} \to [0,1]$ is *negligible* if, for every $c > 0$, there exists an η_0 such that $f(\eta) < \frac{1}{\eta^c}$, for all $\eta > \eta_0$.

5 Security and Accountability of Hashgraph

In this chapter, we present our proofs for Hashgraphs's consistency and accountability properties. We begin with some general discussions about parameters, signatures, and hash collisions. It is convenient to define some general basic correctness conditions that are enforced in a run of the system in order to prove fork-freeness and consistency; we define these conditions in Section 5.1 and prove that Hashgraph is accountable w.r.t. basic correctness. In Section 5.2 we show that Hashgraph is accountable w.r.t. fork-freeness. Then, we present our proof of Hashgraph's consistency in Section 5.3 and show that Hashgraph is accountable w.r.t. consistency in Section 5.4. Lastly, we have a closing discussion in Section 5.5.

In what follows, we always look at runs of the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$ for some environment $\mathcal{E}$ and some adversary $\mathcal{A}$ and require that the Hashgraph protocol $\mathcal{P}^{\mathsf{H}}$ has sound parameters. These requirements are summarized in the next definition.

Definition 3 (Parameters for $\mathcal{P}^{\mathsf{H}}$).
Let ω be a run of the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$ for some environment $\mathcal{E}$ and some adversary $\mathcal{A}$, where $\mathcal{P}^{\mathsf{H}}$ is the Hashgraph protocol as in Definition 1 with parameters

- $\eta \in \mathbb{N}$ *the security parameter,*
- *a finite set of hashgraph node identities* $\mathsf{nodes} \subseteq \{0,1\}^*$,
- *the parameter* $\mathsf{coinRound} \in \mathbb{N}_2$ *that specifies the coin round interval during virtual voting,*
- *a polynomial* $\mathsf{p} \in \mathbb{Z}[x]$ *that limits the runtime of externally provided algorithms,*
- *an EUF-CMA secure signature scheme* $\Sigma = (\mathsf{gen}(1^\eta), \mathsf{sig}, \mathsf{ver})$*, where* gen*,* sig*, and* ver *run for at most* $\mathsf{p}(\eta + |x|)$ *steps on input x,*

and where all internal subprotocols use the same parameters as $\mathcal{P}^{\mathsf{H}}$.

Signatures. We first take a closer look at signatures and define what we understand under a *forged signature*:

Definition 4 (Forged Signature).
Consider a run ω as in Definition 3. For $pid \in \mathsf{nodes}$, the signature σ is a forgery *of message msg for pid if, at some point $\lfloor\omega\rfloor$,*

$$\mathtt{verifySig}_{\mathsf{pk}(pid)}(msg, \sigma) = \mathtt{true}$$

but pid never signed msg at $\mathcal{F}_{\mathtt{cert}}$ up to point $\lfloor\omega\rfloor$.

Recall from Section 3.2.1 that the adversary cannot simple forge valid signatures of another honest node pid (in session sid) by calling the entity $e = (pid, (pid, sid, \mathcal{P}^{\mathsf{H}}_{\mathtt{node}} : \mathtt{node}), \mathtt{signer})$ via a corrupted node, due to the specification of $\mathbf{AllowAdvMessage}$. Therefore, $\mathcal{A}$ cannot forge valid singatures for pid if its signing entity e is uncorrupted; that is, $\mathcal{A}$ has not send the message `corruptSigKey` to e on the network interface beforehand to corrupt the signing key of pid in session sid.

In the following, we often have statements that are only valid because signatures cannot be forged. We will (often) denote such statements at the end with (*).

Hashes. The foundation of all subsequent proofs is the fact that hashes can only be forged with negligible probability:

Lemma 1 (Hash Collisions).
For a run ω as in Definition 3 holds true that $\mathcal{A}$ can only find two $x, x' \subseteq \{0,1\}^*$ such that an instance of $\mathcal{F}_{\mathtt{ro}}$ returns for both x and x' the same hash with negligible probability in η.

Proof. By definition of $\mathcal{F}_{\mathtt{ro}}$, the random oracle outputs hashes with length η. Since the system runs in polynomial time, the probability of finding a collision is negligible in η [10]. ■

The above lemma is pivotal to ensure basic structural conditions on the hashgraph of honest nodes (e.g. self-parent and other-parent of events should be uniquely indentifiable). In particular, this is necessary to ensure that honest nodes have consistent hashgraphs (i.e., they share the same subgraph of all events in common). This will be later discussed in Sections 5.3 and 5.4.

Similar to signatures, we often have statements that are generally true only if hash collisions cannot be found. Since hash collision can be found with negligible probability, we will (often) mark such (*"mostly true"*) statements with (*).

5.1 Accountability of Hashgraph w.r.t. Basic Correctness

Throughout this chapter we will consider only a single hashgraph instance. Since hashes and signatures cannot be forged (except for negligible probability), one can conclude at the end of this chapter that an honest participant cannot be unrightfully blamed (except for negligible probability) due to interference of another hashgraph session.

Definition 5 (Basic Graph Correctness).
Let nodes *be the set of Hashgraph node identities (as specified as a parameter for $\mathcal{P}^{\mathsf{H}}$). Let G_i be the local hashgraph of a node $pid_i \in$* nodes*. We say that hashgraph G_i* fulfills basic graph correctness *if all the below conditions are satisfied:*

1. (complete parent-construction) *For each event $E \in G_i$, different to the genesis event, exist events $E', E'' \in G$ such that*
   (i) (complete self-ancestor) $pid = pid'$ *and* $selfParent = eventID'$*, and*
   (ii) (complete other-ancestor) $pid \neq pid''$ *and* $otherParent = eventID''$.

2. (signature validity) *All events $E \in G_i$, where E is not the genesis event, have a valid signature, i.e., the verification algorithm outputs*

$$\texttt{verifySig}_{\mathsf{pk}(pid)}((\mathit{selfParent}, \mathit{otherParent}, \mathit{transactions}, \mathit{ts}), \sigma) = \texttt{true}.$$

3. (hash validity) *For all events $E \in G_i$, where E is not the genesis event applies*

$$\mathit{eventID} = \texttt{hash}((\mathit{selfParent}, \mathit{otherParent}, \mathit{transactions}, \mathit{ts})).$$

4. (unambigous root) *In G_i exists exactly one event with indegree[1] 0, and for each node $pid \in$ nodes exists at most one event E' such that $\mathit{selfParent}' = \bot$, i.e., the genesis event is the $\mathit{selfParent}$ of E'.*

5. (role compliance) *For all $E \in G_i$, where E was signed by pid applies $pid \in$ nodes, i.e., the creator of E is a node participant.*

Definition 6 (Accountability w.r.t. Basic Corretness).
Consider a run ω as in Definition 3. Let J be an instance of $\mathcal{F}^{\mathsf{H}}_{\texttt{judge}}$ and let nodes *denote the set of all Hashgraph node identities. Let the* witness set W *denote the set of all collected evidences, of form $(\mathit{ctr}, \mathit{pid}, G(\mathit{pid}))$, by J for all $pid \in$ nodes until $\lfloor \omega \rfloor$; formally,*

$$\mathsf{W} \subseteq \mathbb{N} \times \mathsf{nodes} \times ((\mathsf{nodes} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*) \cup (\{\bot\}^7)).$$

This set stores all reported hashgraphs, with well-formed data (see below), of pid with ctr depicting the amount of hashgraphs that were previously submitted by any party to J. We now define the following three security properties of the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$:

$\boldsymbol{\alpha_1}$ ***(Well-Formed Data).*** *The security property α_1 includes all runs ω in which any of the succeeding condition is violated:*

(i) There exists no event $E \in G(pid)$, different to the genesis event, which was reported to J by pid that is not of form $(pid,\ \mathit{eventID},\ \mathit{selfParent},\ \mathit{otherParent},\ \mathit{transactions},\ \mathit{ts},\ \sigma)$ with $pid \in$ nodes, $\mathit{eventID}, \mathit{selfParent}, \mathit{otherParent} \in \{0,1\}^\eta$, $\mathit{transactions}, \sigma \in \{0,1\}^*$, and $ts \in \mathbb{N}$.

(ii) No client submitted a state update $((pid, \mathit{msglist}), \sigma)$ to J such that $pid \in$ nodes and $\texttt{verifySig}_{\mathsf{pk}(pid)}(\mathit{msglist}, \sigma) = \texttt{true}$ but $\mathit{msglist}$ is not a subset of $\mathbb{N} \times \{0,1\}^\eta \times$ $\times \{0,1\}^\eta \times \{0,1\}^*$.

$\boldsymbol{\alpha_2}$ ***(Basic Graph Correctness).*** *Let $\omega \in \alpha_2$ if $\omega \notin \alpha_1$ and at some point during the run ω there exists an entry $(\mathit{ctr}, \mathit{pid}, G(\mathit{pid})) \in \mathsf{W}$, such that the hashgraph G of pid violates any of the basic graph correctness properties defined in Definition 5 (Therefore, α_1 includes all runs in which $\omega \notin \alpha_1$ and basic graph correctness was violated in a hashgraph that was submitted to J).*

[1] In graph theory, *indegree* is the number of edges directed into a vertex in a directed graph. In terms of hashgraph, each event has indegree 2 except for the genesis event $(\bot, \bot, \bot, \bot, \bot, \bot, \bot)$, which has indegree 0.

α_3 *(Valid Evidence).* *We define α_3 to include all runs ω, where $\omega \notin \alpha_1 \cup \alpha_2$ and a party $pid \in \mathsf{nodes}$ submitted a hashgraph that violates the following condition: For each submitted hashgraph G from pid to J applies $G' \subsetneq G$ for all entries $(ctr, pid, G') \in \mathsf{W}$ (i.e., each submitted hashgraph G by pid is a proper superset of each previously submitted hashgraph G' by pid.).*

α_4 *(Valid Msglist).* *Let $msglist$ be the message list that was signed by $pid \in \mathsf{nodes}$ and submitted to J by a client with a valid signature. The security property α_4 comprises all runs ω that are not in $\bigcup_{i=1}^{3} \alpha_i$ and where at least one of the following conditions is violated:*

(i) For each entry $(i, txId, eventID, msg) \in msglist$ applies $txId = \mathtt{hash}(msg)$ (i.e., nodes are obligated to compute valid hashes (i.e., transaction IDs) for all messages they include in their msglist).

(ii) There exists exactly one entry $(i, txId, eventID, msg) \in msglist$ for all $i = 1, \ldots, |msglist|$ (i.e., the reported msglist from pid has to be a consecutive sequence with no duplicates or gaps w.r.t. the first parameter i).

Lastly, we define with $\alpha = \bigcup_{i=1}^{4} \alpha_i$ the accountability constraint C_1^{H} as follows:

$$C_1^{\mathsf{H}} := (\alpha \Rightarrow \mathsf{dis}(pid_1) \mid \cdots \mid \mathsf{dis}(pid_n)). \tag{5.1}$$

As discussed in Example 2, this constraint ensures in a violation of α that J blames at least one individual from the set nodes*. Further, we set the accountability property $\Phi_1 := \{C_1^{\mathsf{H}}\}$.*

We say $\mathcal{P}^{\mathsf{H}}$ with parameters as in Definition 3 is individually accountable w.r.t. basic correctness *if for all environments $\mathcal{E}$ and adversaries $\mathcal{A}$ the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$ is Φ_1-accountable w.r.t. J.*

Lemma 2 (Hashgraph achieves accountability w.r.t. basic correctness).
Consider runs for the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$ with parameters for $\mathcal{P}^{\mathsf{H}}$ as in Definition 3 and let J be an instance of $\mathcal{F}_{\mathtt{judge}}^{\mathsf{H}}$. Then, it holds true that $\mathcal{P}^{\mathsf{H}}$ is individually accountable w.r.t. basic correctness.

Proof. Let $\mathcal{P}^{\mathsf{H}}$ be the Hashgraph protocol with parameters from the lemma above, and let $\mathcal{E}$ be an arbitrary environment and $\mathcal{A}$ be an arbitrary adversary. Towards proving this lemma, we have to show that J ensures Φ_1-accountability for the system $\mathbb{Q} = \{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$; we do this by proving fairness and completeness independently. Before doing this, we first discuss some general observations:

Handling of state updates in $\mathcal{F}_{\mathtt{judge}}^{\mathsf{H}}$ with invalid signatures
In Figure 3.4 nodes sign their current state, the tuple $(pid, currentState)$, with their own signature key, i.e., nodes calculate the signature $\sigma = \mathtt{sign}_{\mathsf{sk}(pid)}((pid, currentState))$ before sending a state update, $(pid, currentState)$ together with the signature σ, to NET (which may be received as parameter $((pid_{send}, msglist), \sigma)$ by a machine instance of $\mathcal{P}_{\mathtt{client}}^{\mathsf{H}}$ (cf. Figure 3.2) or $\mathcal{F}_{\mathtt{judge}}^{\mathsf{H}}$ (cf. Figure 3.14). The signature ensures that only misbehaving nodes are blamed by J (*): Upon receiving a state update $((pid_{send}, msglist), \sigma)$, J first checks whether σ is indeed a valid signature of the tuple. If this check fails, J discards the state update, since J cannot decide which party, if any, misbehaved. There are multiple scenarios which can lead to this occurrence of invalid signatures:

1. pid_{send} is dishonest and included an invalid signature in its state update that was then reported to J by a dishonest client (By implementation of $\mathcal{P}^{\mathsf{H}}_{\mathtt{client}}$, honest clients only submit state updates to an instance of $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$ iff the signature contained in the state update is valid, i.e., $\mathtt{verifySig}_{\mathsf{pk}(pid_{send})}((pid_{send}, currentState), \sigma)$ evaluates to $\mathtt{true}$).
2. pid_{send} is honest but a dishonest client reported malformed evidence (e.g., an invalid signature or message list) to J such that $\mathtt{verifySig}_{\mathsf{pk}(pid_{send})}((pid_{send}, currentState), \sigma) = \mathtt{false}$.

This is not a complete enumeration of all possible scenarios (for instance, the attacker $\mathcal{A}$ can also modify a state update, since all communication between clients and nodes is via network interfaces), however, both scenarios illustrate that pid can be both honest and dishonest. Thus, to achieve fairness, J does not state any verdicts for violations w.r.t. a message list from a state update containing an invalid signature.

Verdicts of J w.r.t. basic correctness
Notice that all verdicts of J w.r.t. basic correctness (see Definition 6) are of form $\mathsf{dis}(\mathsf{pid_{call}})$, $\mathsf{dis}(pid_{send})$, or $\mathsf{dis}(pid)$, where $\mathsf{pid_{call}}$, pid_{send}, and pid has to be party IDs included in the set nodes. Clearly, these verdicts achieve individual accountability for the accountabillity constraint C^{H}_1.

Fairness. To prove fairness, one has to show that $\Pr[\mathbb{Q}(1^\eta) \mapsto \{(J : \psi) \mid \lfloor\omega\rfloor \not\models \psi\}]$ is negligible as a function in η. We do this by showing that J outputs a verdict ψ in ω which evaluates to false at point $\lfloor\omega\rfloor$ (written $\lfloor\omega\rfloor \not\models \psi$) in at most negligible amount of runs $\omega \in \alpha$. Subsequently, we show for each run in α_i, $i \in \{1, 2, 3, 4\}$, that J never renders a wrong verdict (except for negligible amount of runs), and thus conclude that J never outputs a false verdict w.r.t. α (except for a negligible amount of runs)[2]; this then infers fairness of J. By definition, the judge J checks all properties α_i, for $i \in \{1, \ldots, 4\}$. So, to fulfill fairness, we have to show that honest nodes never report hashgraphs to $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$ and send state updates to $\mathtt{NET}$ that violate basic correctness conditions.

α_1* *(Well-Formed Data). We have to show that honest nodes always provide well-formed data, their local hashgraph, to $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$. Further, message lists in state updates submitted by clients (which they received from nodes via the network interface) must suffice a particular format.

(i) One can observe in Figure 3.4 honest nodes sending their internal hashgraph (the state variable events) to $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$. Since the variable events already has the required format (cf. Figure 3.12), no honest node will be blamed by J.

(ii) Figure 3.4 illustrates the computations of an honest node, say pid, before sending a state update to $\mathtt{NET}$ (and therefore possibly to J): pid extracts all entries of form $(counter, txId, eventID, msg) \in \mathbb{N} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^*$ with $counter \neq \varepsilon$ from the local state variable msglist and adds it to the local set $currentState$. Before nodes send their current state to $\mathtt{NET}$, nodes sign the tuple $(pid, currentState)$ with their own signature key, i.e., nodes calculate the signature $\sigma = \mathtt{sign}_{\mathsf{sk}(pid)}((pid, currentState))$. Finally, pid sends the state update, $(pid, currentState)$ together with the signature σ, to $\mathtt{NET}$ (which may be received as parameter $msglist$ by J in Figure 3.14). Consequently, $msglist = currentState$ is also a

[2]This holds true because negligible functions are closed under addition, i.e., for two negligible function $f, g \colon \mathbb{N} \to [0,1]$ is also the function $x \mapsto f(x) + g(x)$ negligible.

subset of $\mathbb{N} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^*$. In case of interference from a malicious client, the signature of the state update is invalid, i.e. $\mathtt{verifySig}_{\mathsf{pk}(pid)}((pid, currentState), \sigma)$ evaluates to `false` (*). As mentioned above, J discards all state updates with invalid signatures. In both cases, J outputs no verdicts.

$\boldsymbol{\alpha_2}$ ***(Basic Graph Correctness).*** Let W denote the set of all collected evidences by participants up to point $\lfloor \omega \rfloor$, as in Definition 6. One can observe in Figure 3.13 J stating the verdicts $\mathsf{dis}(\mathsf{pid}_{\mathsf{call}})$ for the latest submitted hashgraph of $\mathsf{pid}_{\mathsf{call}}$ for misbehavior defined in basic graph correctness, Definition 5. Therefore, it suffices to show that honest nodes only submit hashgraphs that comply with the conditions for basic graph correctness. We show this result with proof by induction:

Base case
For the base case, we observe in the implementation of $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}}$ that the local hashgraph of an honest node is after initialization equal to $\{(\bot,\bot,\bot,\bot,\bot,\bot,\bot)\}$. This set clearly fulfills all conditions for basic graph correctness.

Induction step
For the induction step, we assume the local hashgraph, the set events in $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}}$, of an honest node pid satisfies all conditions for basic graph correctness. Honest nodes only include new events into their local hashgraph during a hashgraph sync, i.e., the process of one node sending a gossip event together with its local hashgraph to another node. Therefore, we only need to consider the circumstance where an honest node merges another hashgraph into its own. After the merging however, a node creates a new event, containing newly submitted transactions, which also has to satisfy basic graph correctness (particularly complete parent-construction). Upon receiving a gossip event (more precisely the event ID of the gossiped event) $gossipedEvent$ together with the hashgraph $events$ over the network interface, pid first does the same basic correctness checks for $events$ as J does with the submitted hashgraphs of pid. In particular, J checks if all events contain valid signatures of valid nodes (cf. Definition 5 role compliance). If any condition in this check is violated, pid does not merge the received hashgraph into its own and thus events remains unchanged. If the previous check succeeds, pid performs some additional checks to ensure $gossipedEvent$ is indeed in the gossiped hashgraph $events$, the creator of $gossipedEvent$ is not pid itself, and $gossipedEvent$ is the last event created by the initiator of the sync.[3] If all aforementioned checks succeed, pid merges the gossiped hashgraph $events$ into his own hashgraph (events). We denote this newly merged hashgraphs as events', which clearly inherits the basic graph correctness properties from events and $events$. Moreover, pid creates a new event (cf. $eventID_{new}$ in Figure 3.4) with valid signature and hash, with the self-parent being the event ID of the last event created by pid and the other-parent being the gossiped event. This newly created event ensures all conditions for basic graph correctness – particularly complete parent-construction. Lastly, this newly created event is added to the local hashgraph of pid. We conclude, events' still satisfies basic graph correctness (**). This completes the induction step.

$\boldsymbol{\alpha_3}$ ***(Valid Evidence).*** We have to show for α_3 that an honest node, say pid, reports hashgraphs to J that are always proper supersets of all previously submitted hashgraphs by pid.

[3] This last property is not necessary in order to prove fairness for J, but it is still a desirable property we enforce for honest nodes.

By definition of $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}}$, honest nodes always create a new event (with event ID $eventID_{new}$), which is added to the local hashgraph events, before reporting their current hashgraph to $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$ at end of a sync. Thus, honest nodes are never blamed by J.

***α_4* (Valid Msglist).** It has to be shown that the message list extracted from a valid state update $((pid_{send}, msglist), \sigma)$ (i) contains valid hashes (the transaction IDs) for all messages and (ii) $msglist$ is a consecutive sequence with no duplicates or gaps. As discussed before, J discards all state updates from clients where the signature of $(pid_{send}, msglist)$ cannot be verified since J cannot judge if pid_{send} actually misbehaved. Therefore, it remains to prove that honest nodes obey conditions (i) and (ii) before sending their state update to `NET`.

(i) One can observe in the implementation of $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}}$ two occurrences where nodes add new messages to their message list (and therefore compute the corresponding transaction IDs): First, nodes may receive a new message msg on their network interface; then, nodes simply calculate the hash $txId$ for this message and add the entry $(\varepsilon, txId, \varepsilon, msg)$ to their message list. Secondly, nodes can receive new events via a hashgraph sync. In this case, if nodes accept the sync, they calculate the transaction IDs for all messages contained in newly received events and add a new entry for each message to their message list. In both cases, we see that nodes compute the hashes for new messages themself and do not rely on transaction IDs from third parties (e.g., other nodes or clients). Therefore, all hashes of messages in message lists from honest nodes are always guaranteed to be valid.

(ii) We observe that honest nodes include in their message list for a state update (cf. $currentState$ in Figure 3.4) all entries $(ctr, txId, eventID, msg) \in \mathsf{msglist}$, where $ctr \neq \varepsilon$ (we call such entries in what follows *ordered entries*). Consequently, it must be shown that at any time ordered entries in $\mathsf{msglist}$ form a consecutive sequence with no duplicates w.r.t. ctr. By definition of $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}}$, nodes add ordered entries only at the end of the function `findOrder` to their message list. Before this occurs, all so far ordered entries $(ctr, txId, eventID, msg) \in \mathsf{msglist}$ are replaced with (unordered) entries $(\varepsilon, txId, eventID, msg)$. Thereafter, nodes iterate over all (now) unordered entries $(\varepsilon, txId, eventID, msg) \in \mathsf{msglist}$, where the event with ID $eventID$ has been asigned a consensus position in the total order of events, and add the ordered entry $(i, txId, eventID, msg)$ to $\mathsf{msglist}$. Thereby, $i = 1$ at the beginning, and i is increased by one each time an entry is added to $\mathsf{msglist}$. Clearly, the resulting message list is a consecutive sequence of ordered entries with no gaps and duplicates.

We conclude, J is fair in all but negligible amount of runs of the system $\mathbb{Q}$.

Completeness. To prove completeness, we have to show that $\Pr[\mathbb{Q}(1^\eta) \mapsto \neg(J : \Phi_1)]$ is negligible as a function in η; that is, J must ensure the accountability constraint C_1^{H} in all but a negligible number of runs ω of the system $\{\mathcal{P}^{\mathsf{H}}, \mathcal{A}, \mathcal{E}\}$ with parameters for $\mathcal{P}^{\mathsf{H}}$ as in Definition 3. Particularly, if ω is contained within the security property α, J has to state a verdict ψ that implies at least one of the verdicts $\mathsf{dis}(pid_1), \ldots, \mathsf{dis}(pid_n)$ defined by C_1^{H} (cf. Example 2). As already argued above, all verdicts of J are of form $\bigwedge_{i \in I} \mathsf{dis}(pid_i)$, for $\emptyset \neq I \subset \{1, \ldots, n\}$; hence, J achieves individual accountability because each verdict rendered by J implies at minimum one of the verdicts defined by the constraint C_1^{H}. In order to prove fairness, we already argued that by definition of $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$, the judge J checks all properties α_i for $i \in \{1, \ldots, 4\}$, i.e., J states a verdict if any of the conditions for basic correctness is violated during a run $\omega \in \alpha$. This can be seen in Figures 3.13 and 3.14.

We conclude this lemma that for a run $\omega \in \alpha$ the judge J never renders a wrong verdict for ω except for a negligible amount of runs, and J always states a verdict for a violation of α. Therefore, we have $\Pr[\mathbb{Q}(1^\eta) \mapsto \{(J : \psi) \mid \lfloor\omega\rfloor \not\models \psi\}]$ is negligible as a function in η, and $\Pr[\mathbb{Q}(1^\eta) \mapsto \neg(J : \Phi_1)] = 0$. This concludes the proof for this lemma. ■

5.2 Accountability of Hashgraph w.r.t. Fork-Freeness

Definition 7 (Fork-Freeness).
Consider a run ω as in Definition 3, where $\omega \notin \alpha$. Let nodes *be the set of Hashgraph node identities (as specified as a parameter for $\mathcal{P}^{\mathsf{H}}$), and assume J is an instance of $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$. Let $\lfloor\omega\rfloor$ denote one point somewhere in the run ω, and let the witness set* W *denote the set of all collected evidences by J of form (i, pid, G) where G, submitted by pid, is the i-th hashgraph that was reported to J by any node prior to $\lfloor\omega\rfloor$ (see Definition 6). We define*

$$\mathbb{E}_{\mathsf{W}} := \bigcup_{(i,pid,G)\in\mathsf{W}} G$$

to be the set of all so far reported events by any node.

Fork-Freeness. *We say $pid \in$* nodes satisfies fork-freeness at point $\lfloor\omega\rfloor$ *if for all pairs of events $E_1, E_2 \in \mathbb{E}_{\mathsf{W}}$ signed by pid it holds true that $selfParent_1 = selfParent_2$ implies $eventID_1 = eventID_2$.*

The system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$ satisfies fork-freeness in the run ω *if there is no $pid \in$* nodes *that violates fork-freeness at some point during the run ω.*

Definition 8 (Accountability w.r.t. Fork-Freeness).
Consider a run ω as in Definition 3. Let nodes *be the set of all Hashgraph node identities, and assume J is an instance of $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$. Let* W *denote the witness set (i.e., the set of all collected evidences of nodes, see Definition 6).*

β_1 (Fork-Freeness). *This security property contains all runs, where $\omega \notin \alpha$ and there exists a node $pid \in$* nodes *such that pid does not satisfy fork-freeness at some point $\lfloor\omega\rfloor$.*

We set $\beta = \beta_1 \cup \alpha$, and define the accountability constraint C^{H}_2 as follows:

$$C^{\mathsf{H}}_2 := (\beta \Rightarrow \mathsf{dis}(pid_1) \mid \cdots \mid \mathsf{dis}(pid_n)). \tag{5.2}$$

This accountability constraint also ensures individually accountability. Finally, we set the accountability property $\Phi_2 := \{C^{\mathsf{H}}_2\}$.

We call $\mathcal{P}^{\mathsf{H}}$ with parameters as in Definition 3 individually accountable w.r.t. fork-freeness *if for all environments $\mathcal{E}$ and adversaries $\mathcal{A}$ the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$ is Φ_2-accountable w.r.t. J.*

Theorem 1 (Hashgraph achieves accountability w.r.t. fork-freeness).
Consider runs for the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$ with parameters for $\mathcal{P}^{\mathsf{H}}$ as in Definition 3 and let J be an instance of $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$. Then, it holds true that $\mathcal{P}^{\mathsf{H}}$ is individually accountable w.r.t. fork-freeness.

In what follows, we present our proof for Theorem 1, i.e., we demonstrate that Hashgraph does achieve individual accountability w.r.t. fork-freeness for runs as in Definition 3.

Proof. Consider a run ω as in Definition 3, where $\mathcal{E}$ is an arbitrary environment and $\mathcal{A}$ is an arbitrary adversary. Further, let Φ_2 be the accountability constraint from Definition 8. We show that the system $\mathbb{Q} = \{\mathcal{P}^{\mathsf{H}}, \mathcal{A}, \mathcal{E}\}$ in the run ω satisfies Φ_2-accountability, i.e., $\{\mathcal{P}^{\mathsf{H}}, \mathcal{A}, \mathcal{E}\}$ fulfills both fairness and completeness.

Fairness. In order to prove fairness, we have to show that $\Pr[\mathbb{Q}(1^\eta) \mapsto \{(J : \psi) \mid \lfloor\omega\rfloor \not\models \psi\}]$ is negligible as a function in η. That is, we have to show that the judge J renders verdicts ψ that evaluate to false at point $\lfloor\omega\rfloor$ (written $\lfloor\omega\rfloor \not\models \psi$) in at most negligible amount of runs of the system $\{\mathcal{P}^{\mathsf{H}}, \mathcal{A}, \mathcal{E}\}$ with parameters as in Definition 3. We have already shown that J is fair for verdicts w.r.t. basic correctness (α), i.e., if $\omega \notin \alpha$, J will not render verdicts for honest nodes except for negligible probability in η. Therefore, it is left to prove that J's verdicts w.r.t. β_1 are also fair:

First, one can observe in Figure 3.13 that J does the exact same checks as specified in the definition of fork-freeness before it outputs a verdict, blaming some participant: Initially, J calculates $\mathbb{E}_{\mathsf{W}}$, the set of all so far reported events by any node, then J checks if there exists a pair of two events $E_1, E_2 \in \mathbb{E}_{\mathsf{W}}$ created by pid for which applies $\mathit{selfParent}_1 = \mathit{selfParent}_2$ and $\mathit{eventID}_1 \neq \mathit{eventID}_2$. Thereafter, J ensures that the signatures σ_1, σ_2 for E_1 and E_2 are indeed valid, i.e., J verifies that $\mathtt{verifySig}_{\mathsf{pk}(pid)}((\mathit{selfParent}_i, \mathit{otherParent}_i, \mathit{txs}_i, \mathit{ts}_i), \sigma_i)$ outputs $\mathtt{true}$ for $i \in \{1, 2\}$. If this check also succeeds, J states the verdict $\mathsf{dis}(pid)$ for node pid. It remains to be shown that pid is indeed dishonest. Since J verifies the signature of two forking events E_1, E_2 created by pid, node pid must have created and signed both events (*). Thus, for J to be fair, honest nodes must not create forked events. By definition of $\mathcal{P}^{\mathsf{H}}_{\mathtt{node}}$, nodes create new events only when they receive and accept a sync from another node. Let E_{new} be such a new event created by pid (i.e., pid signed it) with self-parent being the last event that pid created before E_{new}, or the genesis event in case E_{new} is the first event that pid created. Therefore, each new event created by pid has a different self-parent (⋆); thus, honest nodes never fork (⋆). We conclude that J is fair, since J never renders verdicts, blaming honest nodes, w.r.t. fork-freeness (⋆⋆); hence, $\Pr[\mathbb{Q}(1^\eta) \mapsto \{(J : \psi) \mid \lfloor\omega\rfloor \not\models \psi\}]$ is negligible in η.

Completeness. To prove completeness, one has to show that $\Pr[\mathbb{Q}(1^\eta) \mapsto \neg(J : \Phi_2)]$ is negligible as a function in η, i.e., J ensures the accountability constraint C_2^{H} in all but a negligible number of runs ω of the system $\{\mathcal{P}^{\mathsf{H}}, \mathcal{A}, \mathcal{E}\}$ with parameters as in Definition 3. Particularly, if the run ω is contained within the security property $\beta_1 \cup \alpha$, J has to state a verdict ψ that implies at least one of the verdicts $\mathsf{dis}(pid_1), \ldots, \mathsf{dis}(pid_n)$ defined by C_2^{H}. First, oberserve that all verdicts of J are of form $\bigwedge_{i \in I} \mathsf{dis}(pid_i)$, for $\emptyset \neq I \subset \{1, \ldots, n\}$ (notice that J can state multiple verdicts for different parties); consequently, each such verdict clearly achieves individual accountability for C_2^{H} as each verdict implies at minimum one of the verdicts defined by the constraint C_2^{H}. Since we have already shown J fulfilling completeness for basic correctness (i.e., the run ω is in α), it remains to prove that J is complete w.r.t. fork-freeness (i.e., $\omega \in \beta_1$).

In the following, let ω be a run of β_1; therefore, there exists at some point $\lfloor\omega\rfloor$ in the set $\mathbb{E}_{\mathsf{W}} = \{E \in G \mid (\mathit{ctr}, pid, G) \in \mathsf{W}\}$ a pair of events that are both signed by some node pid, such that pid does not satisfy fork-freeness. More specifically, there are two events $E_1, E_2 \in \mathbb{E}_{\mathsf{W}}$ with $\mathtt{verifySig}_{\mathsf{pk}(pid)}((\mathit{selfParent}_i, \mathit{otherParent}_i, \mathit{txs}_i, \mathit{ts}_i), \sigma_i) = \mathtt{true}$ for $i \in \{1, 2\}$, so that

$selfParent_1 = selfParent_2$ but $eventID_1 \neq eventID_2$, i.e., pid created a fork at $selfParent_1$ with $eventID_1$ and $eventID_2$. One can observe in Figure 3.13 that the judge, indeed, outputs a verdict $\mathsf{dis}(pid)$ for this violation. We conclude that J always renders a verdict in β_1, i.e., $\Pr[\mathbb{Q}(1^\eta) \mapsto \neg(J : \Phi_2)] = 0$.

We have shown that J is fair and complete in all runs of $\{\mathcal{P}^{\mathsf{H}}, \mathcal{A}, \mathcal{E}\}$ with parameters as in Definition 3 (**). This concludes the proof for Theorem 1. ■

5.3 Consistency of the Hashgraph Algorithm

Definition 9 (Consistent Hashgraphs).
Let G be a hashgraph and $E \in G$. Let $G[E] \subseteq G$ denote the subgraph of E in G that contains E and all ancestors of E. We say two hashgraphs G and G' are consistent if for all $E \in G \cap G'$ follows $G[E] = G'[E]$.

Lemma 3.
Let G, G' be two hashgraphs, both satisfying basic graph correctness. Then, it holds true that G and G' are consistent .

Proof. We first prove for an arbitrary event $E \in G \cap G'$, different to the genesis event, that both its parent events are in $G \cap G'$, and there exist no other events in G or G' with the same event ID as one of E's parents events except for negligible probability in η. Because G, G' satisfy basic graph correctness, there must be self-parents $E'_1 \in G$, $E'_2 \in G'$ with $pid = pid'_1$, $pid = pid'_2$ and $selfParent = eventID'_1$, $selfParent = eventID'_2$, and other-parents $E''_1 \in G$, $E''_2 \in G'$ where $pid \neq pid''_1$, $pid \neq pid''_2$ and $otherParent = eventID''_1$, $otherParent = eventID''_2$. Furthermore, hash collisions cannot be found except for negligible probability in η; therefore, there cannot be two events in $G \cup G'$ with the same event ID (*); thus, $E'_1 = E'_2, E''_1 = E''_2$ and self-parent and other-parent of E are unique in G and G' (*).

Recall from Definition 9 that G, G' are consistent if for all $E \in G \cap G'$ follows $G[E] = G'[E]$. We know the genesis event is in G and G', and for each other event $E \in G \cap G'$ there is exactly one self-parent event and exactly one other-parent event in $G \cap G'$. By this, we conclude $G[E] = G'[E]$ (*). ■

In the following, we will usually demand for two hashgraphs G, G' that both fulfill basic graph correctness and follow implicitly with the above lemma that G, G' are also consistent.

Lemma 4.
Let G and G' be two hashgraphs, both satisfying basic graph correctness. Then, it holds true that the `divideRounds` *algorithm assigns for all $E \in G \cap G'$ the same round created number and witness status, when run on input G or G'.*

Proof. Let $E \in G \cap G'$ be arbitrary. Because G and G' are also consistent, by Definition 9 applies $G[E] = G'[E]$, i.e., both hashgraphs share the same subgraph of E. We show this lemma with proof by induction over the subgraph $G[E]$.

Base case
Initially, the genesis event receives round created number 1 and is assigned to be the sole witness of round 1. Additionally, each event E containing the genesis event as self-parent receives round number 2 and is assigned to be a witness of this round; basic graph correctness ensures that there is at most one such event for each creator of E.

Induction step
Assume the induction hypothesis that for an event $E_0 \in G[E] = G'[E]$ it is true that `divideRounds` assigned for all ancestors of E_0, in particular the self-parent E_1 and other-parent E_2, the same round created number when run on hashgraph G or G'. Then, with input G and G', `divideRounds` will assign E_0's round r to be the maxium of the rounds of E_1 and E_2, and if E_0 strongly sees more than $\frac{2}{3}n$ round r witnesses, it will instead assign $r+1$ as round created number of E_0 and mark E_0 as witness of round $r+1$.

Therefore, `divideRounds` assigns on input G and G' for each event in $G[E]$, including E, the same round and witness status. ■

Corollary 1 (Consistent Witnesses).
Let G, G' be two hashgraphs, both satisfying basic graph correctness, with $G \subseteq G'$. Then it holds true that $\mathcal{W}_r \subseteq \mathcal{W}'_r$, i.e., if $E_r \in \mathcal{W}_r$ is a round r witness in G, then E_r is also a round r witness in G'.

Proof. By Lemma 4, for all $E_r \in G \cap G'$ follows $E_r \in \mathcal{W}_r, \mathcal{W}'_r$ or $E_r \notin \mathcal{W}_r, \mathcal{W}'_r$; thus, with $G \subseteq G'$ holds true that $\mathcal{W}_r \subseteq \mathcal{W}'_r$. ■

Lemma 5.
Let G be a hashgraph fulfilling basic graph correctness, and let $\mathcal{W}_r$ denote the set of all round r witnesses in G. If there exists an event $E_r \in \mathcal{W}_r$ whose fame is decided (by running the algorithm `decideFame`*) by a witness $E_{r+i+1} \in \mathcal{W}_{r+i+1}$ in round $r+i+1$ ($i \geq 1$), then it holds true for all $k \in \{1, \ldots, i\}$ that there exist more than $\frac{2}{3}n$ witnesses in $\mathcal{W}_{r+k}$.*

Proof. Let $E_r \in \mathcal{W}_r$ be an event whose fame has been decided during `decideFame`. Therefore, virtual voting of E_r must have at least reached the second round $r+2$, otherwise $E'_r s$ fame could not have been decided (witnesses in the first round do not decide the fame for the candidate); hence, $i \geq 1$ for witness E_{r+i+1} which decides the fame for E_r. Suppose $E_{r+i+1} \in \mathcal{W}_{r+i+1}$ is a witness that decides the famousness of E_r. One can observe in `decideFame` that E_{r+i+1}'s decision depends solely on the round $r+i$ witnesses $E^1_{r+i}, \ldots E^m_{r+i}$ it strongly sees, where $m > \frac{2}{3}n$ by the definition of witnesses. If $i = 1$, then $|\mathcal{W}_{r+1}| \geq m > \frac{2}{3}n$ and the above statement holds true.

Suppose $i > 1$. A witness $E_{r+k+1} \in \mathcal{W}_{r+k+1}$ *participates in the virtual voting* for E_r, if $k = i$, or $k \geq 1$ and there exist $l = i - k$ witnesses $\widehat{E}_{r+k+2} \in \mathcal{W}_{r+k+2}, \ldots, \widehat{E}_{r+i+1} \in \mathcal{W}_{r+i+1}$ such that $\widehat{E}_{r+p+1}$ strongly sees $\widehat{E}_{r+p}$ for all $p \in \{k+2, \ldots, i\}$, and $\widehat{E}_{r+k+2}$ strongly sees E_{r+k+1}. A participating witness E_{r+k+1} strongly sees at least least $m_k > \frac{2}{3}n$ witnesses $E^1_{r+k}, \ldots, E^{m_k}_{r+k} \in \mathcal{W}_{r+k}$ in round $r+k$, which also participate in the election if $k \geq 2$. Since E_{r+k+i} participates in the election, we inductively conclude that there must exist at least $m_k > \frac{2}{3}n$ witnesses $E^1_{r+k}, \ldots, E^{m_k}_{r+k} \in \mathcal{W}_{r+k}$ in round $r+k$ for all $k \in \{1, \ldots, i\}$; thus, $|\mathcal{W}_{r+k}| \geq m_k > \frac{3}{2}n$. ■

Lemma 6 (Consistent Voting).
Let G, G' be two hashgraphs such that $G \subseteq G'$ and both satisfy basic graph correctness. Then, for all $E_r \in \mathcal{W}_r$ and $E_{r+i} \in \mathcal{W}_{r+i}$ with $\mathsf{vote}_G(E_r, E_{r+i}, \beta)$ *(where $\beta \in \{\texttt{true}, \texttt{false}\}$) holds true that* $\mathsf{vote}_{G'}(E_r, E_{r+i}, \beta)$*, if there is no $E_{r+i'+1} \in \mathcal{W}'_{r+i'+1}$ with* $\mathsf{decision}_{G'}(E_r, E_{r+i'+1}, \beta')$ *that votes before E_{r+i} in G'. Moreover, if additionally applies* $\mathsf{decision}_G(E_r, E_{r+i}, \beta)$*, then it holds true that* $\mathsf{decision}_{G'}(E_r, E_{r+i}, \beta)$ *if there exists no witnesses that ends E_r's election (see above) before E_{r+i} can vote.*

Proof. Observe that voting (and deciding the fame for some witness) is purely a function on the ancestors of voting (deciding) witnesses. Because of $G \subseteq G'$, the ancestors for events in G, clearly, to do not change in G' (note that $G \subseteq G'$ implies that G, G' are consistent hashgraphs). Thus, events will vote (decide) identically for E_r in G' if the election for E_r is not ended beforehand by some new witness $E_{r+i'+1} \in \mathcal{W}'_{r+i'+1} \setminus \mathcal{W}_{r+i'+1}$. ■

The purpose of strongly seeing is to ensure that nodes with different hashgraphs achieve consistent voting results. We prove in the following lemma that this holds true as long as more than $\frac{2}{3}n$ nodes do not create forks. This result is the foundation for Hashgraph to achieve consistency, and will be used frequently for the rest of the remaining proofs.

Lemma 7 (Strongly Seeing Lemma).
Let G, G' be two hashgraphs with $G \subseteq G'$ and both fulfilling basic graph correctness. Suppose G' contains less than $\frac{1}{3}n$ forking nodes. Let the pair of events (E_1, E_2) be a fork with self-parent E_0 in G. If E_1 is strongly seen by event E in G, then E_2 will not be strongly seen by all events in G' (cf. [2][4]).

Proof. We prove that no event in G' can strongly see E_2. Assume E_1 is strongly seen by some event $E \in G$. By the definition of strongly seeing, there must exist a set of events $S_1 \subseteq G$ such that $|S_1| > \frac{2}{3}n$, all events in S_1 have pairwise different creators, and for all $E_s \in S_1$ applies E sees E_s and E_s sees E_1. Let $\mathcal{N}_1 = \{\mathsf{creator}_G(E_s) \mid E_s \in S_1\} \subseteq \mathsf{nodes}$ be the set of all creators of events in S_1, and let $\mathcal{N}_1^H \subseteq \mathcal{N}_1$ denote the set of all honest nodes (i.e., nodes without forks) in $\mathcal{N}_1$. Because of the assumption of less than $\frac{1}{3}n$ forking nodes, it holds true that $|\mathcal{N}_1^H| > \frac{1}{3}n$.

Let $E' \in G'$ be an arbitrary event; we prove that E' cannot strongly see E_2 in G'. For the purpose of contradiction, let $S_2 \subseteq G'$ be a set of events such that $|S_2| > \frac{2}{3}n$, all events in S_2 have pairwise different creators, and for all events $E_s \in S_2$ holds true that E' sees E_s and E_s sees E_2. We show that this set cannot exist. Again, let $\mathcal{N}_2$ be the set of all creators of events in S_2, where $|\mathcal{N}_2| > \frac{2}{3}n$ by the definition of strongly seeing. Then, we can conclude $|\mathcal{N}_2 \cap \mathcal{N}_1^H| \geq 1$. Therefore, there must be an honest node $pid \in \mathcal{N}_2 \cap \mathcal{N}_1^H$ with no forks in G and G'. Moreover, there must be events $E_1^H \in S_1, E_2^H \in S_2$ created by pid such that E_1^H sees E_1 and E_2^H sees E_2. However, this is not possible: Due to $G \subseteq G'$ and E_1^H seeing E_1 in G, E_1^H can also see E_1 in G'. By the definition of seeing, the subgraph $G[E_1^H] = G'[E_1^H]$ does not contain a fork of E_1; thus $E_2 \notin G'[E_1^H]$. Therefore, $E_1^H \neq E_2^H$ and E_1^H is an self-ancestor of E_2^H. But then is also E_1 an anestor of E_2^H. So both E_1 and E_2 are ancestors of E_2^H; hence, E_2^H cannot see either of two the events. This is a contradiction to $E_2^H \in S_2$. Therefore, S_2 does not exist and E_2 is not strongly seen in G'. ■

[4]The author of this paper proved a slightly stronger statement: Instead of assuming $G \subseteq G'$, Leemon Baird proved this lemma with the more strict requirement of G, G' being consistent hashgraphs.

We proved the above lemma for all hashgraphs G, G that satisfy basic graph correctness, where $G \subseteq G'$. However, we cannot guarantee that hashgraphs of different nodes are always subsets of each other, but rather expect honest nodes to have consistent hashgraphs. We will later demonstrate in Theorem 2 that our assumption of $G \subseteq G'$ is sufficient, and we do not have to prove Lemma 7 under the more strict condition of G, G' being consistent hashgraphs.

Corollary 2.
Let G be a hashgraph satisfying basic graph correctness and suppose less than $\frac{1}{3}n$ forking nodes in G. Let $\mathcal{W} \subseteq \mathcal{W}_r$ be the set of round r witnesses that are strongly seen by events in G. Then it holds true for all pairs of events (E_r, E_r') in $\mathcal{W}$ that $\mathsf{creator}_G(E_r) \neq \mathsf{creator}_G(E_r')$.

Proof. By definition, a witness of round r is the first event created by a node, say pid, in this round. In the absence of forking, there is at most one witness created by pid in round r. If pid created a fork, we know by Lemma 7 that at most one event of the forked round r witnesses of pid is strongly seen in G. Hence, $\mathcal{W}$ contains only round r witnesses of different creators. ■

Corollary 3.
Let G be a hashgraph satisfying basic graph correctness and suppose less than $\frac{1}{3}n$ forking nodes in G. Then it holds true that all witnesses in $\mathcal{W}_{r+1}$ can strongly see at most n witnesses of round r. In particular, there exist at most n witnesses in $\mathcal{W}_r$ that are strongly seen by witnesses in $\mathcal{W}_{r+1}$.

Proof. Follows immediately by Lemma 7 and Corollary 2. ■

Lemma 8 (Consistent Famous Witnesses).
Let G, G' be two hashgraphs such that $G \subseteq G'$ and both satisfy basic graph correctness. Further, suppose G' contains less than $\frac{1}{3}n$ forking nodes. Let $\emptyset \neq \mathcal{W}_r, \mathcal{W}_r'$ be the sets of all known witnesses with round created r in G and G', respectively. Let $\mathcal{F}_r \subseteq \mathcal{W}_r$ and $\mathcal{F}_r' \subseteq \mathcal{W}_r'$ be the sets of all famous witnesses of round r.

If the fame state for all witnesses in $\mathcal{W}_r$ is decided, then it holds true that $\mathcal{F}_r = \mathcal{F}_r'$, i.e., it is not possible to discover new famous witnesses of round created r or alter the fame state of round r witnesses once the fame of all round r witnesses is decided.

Proof. Let all variables be as specified above. Assume that the famousness for all witnesses in $\mathcal{W}_r$ is decided, i.e., each event in $\mathcal{W}_r$ is either famous or not. Moreover, we can assume by Corollary 1 that $\mathcal{W}_r \subseteq \mathcal{W}_r'$ for all $r \in \mathbb{N}$.

Proof of $\mathcal{F}_r \subseteq \mathcal{F}_r'$
Let $E_r \in \mathcal{W}_r$ be an arbitrary but fixed witness whose fame has been decided by some $E_{r+i+1} \in \mathcal{W}_r$, written $\mathsf{decision}_G(E_r, E_{r+i+1}, \beta)$, for $i \geq 1$ with decision $\beta \in \{\texttt{true}, \texttt{false}\}$. We prove the existence of a witness $E'_{r+i'+1} \in \mathcal{W}'_{r+i'+1}$ with $\mathsf{decision}_{G'}(E_r, E'_{r+i'+1}, \beta')$ such that $\beta' = \beta$; thus, we show a stronger statement: We do not only prove that (1) $E_r \in \mathcal{F}_r$ implies $E_r \in \mathcal{F}_r'$, but also the implication (2) $E_r \notin \mathcal{F}_r \Rightarrow E_r \notin \mathcal{F}_r'$. In the what follows, we present a direct proof of both implications.

By Lemma 6, we know that $\mathsf{decision}_{G'}(E_r, E_{r+i+1}, \beta)$, if there exists no witness that ends E_r's virtual voting (by deciding its fame) before E_{r+i+1} can vote. First, assume this is the case; thus $E'_{r+i'+1} = E_{r+i+1}$ and (1) and (2) holds true. So, we only have to prove that (1) and (2) also hold true if $E'_{r+i'+1} \neq E_{r+i+1}$, which implies that $E'_{r+i'+1}$ ends E_r's election by deciding its fame before E_{r+i+1} has the opportunity to do so. Clearly, this is only the case if $i' \leq i$, which we assume in the following. Also notice that $i' \geq 1$ (cf. proof of Lemma 5), and $E_{r+i'+1} \in G' \setminus G$ by Lemma 6.

$E'_{r+i'+1}$ decides the fame for E_r w.r.t. the supermajority decision of the witnesses in the previous round that $E'_{r+i'+1}$ can strongly see; let $\mathcal{W}' \subseteq \mathcal{W}'_{r+i'}$ be the set of such witnesses that $E'_{r+i'+1}$ strongly sees. By Corollary 3 and the definition of witnesses, it must be $\frac{2}{3}n < m' = |\mathcal{W}'| \leq n$, and the witnesses in $\mathcal{W}'$ have pairwise different creators by Corollary 2. We further define $\mathcal{W}'_{\beta'} \subseteq \mathcal{W}'$ to be the set of all witnesses in round $r + i'$ with a vote of β', where $|\mathcal{W}'_{\beta'}| > \frac{2}{3}n$ since $E_{r+i'+1}$ has a supermajority of votes of β'. Next, we define the *voting set* of round $r + i'$,

$$\mathcal{V}_{r+i'} = \{E_{r+i'} \in \mathcal{W}'_{r+i'} \mid \exists E_{r+i'+1} \in \mathcal{W}'_{r+i'+1} \colon E_{r+i'+1} \textit{ strongly sees } E_{r+i'}\}, \tag{5.3}$$

to be the set of all witnesses of round $r + i'$ in G' that are strongly seen by some witness of round $r + i' + 1$ in G', where $|\mathcal{V}_{r+i'}| \leq n$ by Corollary 3. With Corollary 2 we can conclude that all pairs of witnesses in $\mathcal{V}_{r+i'}$ have different creators. Subsequently, we differentiate whether *(i)* $i' = i$ or *(ii)* $i' < i$.

Case (i)
Suppose $i' = i$ first. Then there must exist a set $\mathcal{W} \subseteq \mathcal{W}_{r+i}$ (cf. Lemma 5) with $|\mathcal{W}| > \frac{2}{3}n$ that E_{r+i+1} can strongly see, where each pair of witnesses in $\mathcal{W}$ have pairwise different creators (see Corollary 2). Additionally, we conclude with Corollary 3 that $\frac{2}{3}n < |\mathcal{W}| \leq n$. Because of $\mathsf{decision}_G(E_r, E_{r+i+1}, \beta)$, we have for all E_{r+i} with $\mathsf{vote}_G(E_r, E_{r+i}, \beta)$ also $\mathsf{vote}_{G'}(E_r, E_{r+i}, \beta)$ by Lemma 6; let $\mathcal{W}_\beta \subseteq \mathcal{W}_{r+i}$ be the set of all such E_{r+i}. Clearly, it applies $|\mathcal{W}_\beta| > \frac{2}{3}n$ and $\mathcal{W}_\beta \subseteq \mathcal{V}_{r+i}$ (cf. Corollary 1), but we also have $\mathcal{W}'_{\beta'} \subseteq \mathcal{V}_{r+i}$. Therefore, $\mathcal{V}_{r+i}$ must have more than $\frac{2}{3}n$ witnesses with votes for β and β'. With $|\mathcal{V}_{r+i}| \leq n$, we conclude that $\beta = \beta'$; thus (1) and (2) are satisfied.

Case (ii)
Suppose $i' < i$. This case is a little more sophisticated. We claim the existence of at least $\frac{1}{3}n$ witnesses $E_{r+i'} \in \mathcal{W}_{r+i'}$ such that $\mathsf{vote}_G(E_r, E_{r+i'}, \beta)$ (and therefore also $\mathsf{vote}_{G'}(E_r, E_{r+i'}, \beta)$ by Lemma 6); let $\mathcal{W}_\beta \subseteq \mathcal{W}_{r+i'}$ denote the set of such witnesses.

We show $|\mathcal{W}_\beta| \geq \frac{1}{3}n$ with proof by contradiction. Suppose for the purpose of contradiction that $|\mathcal{W}_\beta| < \frac{1}{3}n$. Then, all round $r + i' + 1$ witnesses $E_{r+i'+1}$ in G will strongly see at least $\frac{1}{3}n$ witnesses of the previous round with vote $\overline{\beta}$ for E_r; thus, $\mathsf{vote}_G(E_r, E_{r+i'+1}, \overline{\beta})$. Hereafter, we have to distinguish whether round $r + i' + 2$ is a normal voting or coin round. First assume round $r + i' + 2$ is not a coin round. Then all witnesses of this round will have a supermajority of votes $\overline{\beta}$ of the round $r + i' + 1$ witnesses they strongly see. Let $E_{r+i'+2}$ be the first witness in this round that can vote, then $\mathsf{decide}_G(E_r, E_{r+i'+2}, \overline{\beta})$. This is clearly a contradiction to the assumption $\mathsf{decision}_G(E_r, E_{r+i+1}, \beta)$. Oppositely, suppose $r + i' + 2$ is a coin round. Notice that witnesses in coin rounds cannot decide the fame; thus $i' - i \geq 2$. However, if they have a supermajority of votes, as in our case in favour of $\overline{\beta}$, they will still vote according to the supermajority voting decision; hence, $\mathsf{vote}_G(E_r, E_{r+i'+2}, \overline{\beta})$ for all $E_{r+i'+2} \in \mathcal{W}_{r+i'+2}$. Finally, the first witness of round $r + i' + 3$ able to vote will see a supermajority of $\overline{\beta}$ votes; thus $\mathsf{decision}_G(E_r, E_{r+i+3}, \overline{\beta})$, which is again a contradiction to the assumption $\mathsf{decision}_G(E_r, E_{r+i+1}, \beta)$. Concluding, $|\mathcal{W}_\beta| \geq \frac{1}{3}n$.

So, we have $\mathcal{W}_\beta \subseteq \mathcal{W}'_{r+i'} \subseteq \mathcal{V}_{r+i'}$, but also $\mathcal{W}'_{\beta'} \subseteq \mathcal{V}_{r+i'}$. We know $|\mathcal{W}'_{\beta'}| > \frac{2}{3}n$, and with $|\mathcal{V}_{r+i'}| \leq n$ there can be at most $\frac{2}{3}n$ witnesses in $\mathcal{W}'_{r+i'}$ with vote $\overline{\beta}$ for E_r, therefore missing one vote to have a supermajority decision of $\overline{\beta}$ in favor of E_r; thus, $\beta' = \beta$, $\mathsf{decision}_{G'}(E_r, E_{r+i'+1}, \beta)$, and (1), (2) are fulfilled. In particular, we conclude $\mathcal{F}_r \subseteq \mathcal{F}'_r$.

Proof of $\mathcal{F}_r \supseteq \mathcal{F}'_r$

Proof by contrapositive. Let $E'_r \in \mathcal{W}'_r$ be an arbitrary but fixed round r witness that is not in $\mathcal{F}_r$. First, suppose additionally $E'_r \in \mathcal{W}_r$. Then, by implication (2) $E_r \notin \mathcal{F}_r \Rightarrow E_r \notin \mathcal{F}'_r$ from earlier, we know $E'_r \notin \mathcal{F}'_r$. So, let $E'_r \in \mathcal{W}'_r \setminus \mathcal{W}_r$ in the following.

Because of $\mathcal{W}_r \neq \emptyset$, there must exist a witness $E_r \in \mathcal{W}_r$ whose fame has been decided in some round $r + i + 1$ ($i \geq 1$) by some $E_{r+i+1} \in \mathcal{W}_{r+i+1}$, written $\mathsf{decide}_G(E_r, E_{r+i+1}, \beta)$, during `decideFame`. Thus, virtual voting of E_r must have at least reached round $r + 2$. By Lemma 5, there must exist a set of round $r + 1$ witnesses $\mathcal{W} \subseteq \mathcal{W}_{r+1}$, with $|\mathcal{W}| > \frac{2}{3}n$, that are strongly seen by some $E_{r+2} \in \mathcal{W}_{r+2}$ (cf. proof of Lemma 5) that voted for E_r. Because of $E'_r \notin \mathcal{W}_r$, and more generally also $E'_r \notin G$, all round $r + 1$ witnesses in $\mathcal{W} \subseteq G$ cannot be descendants of E'_r. Therefore, $\mathsf{vote}_{G'}(E'_r, E_{r+1}, \texttt{false})$ for all $E_{r+1} \in \mathcal{W}$. Furthermore, we now $\mathcal{W} \subseteq \mathcal{V}_{r+1}$, so there cannot be more than $\frac{1}{3}n - 1$ `true`-votes for E'_r in $\mathcal{V}_{r+1}$; thus, $\mathsf{decide}_{G'}(E'_r, E_{r+2}, \texttt{false})$ or $\mathsf{decide}_{G'}(E'_r, E'_{r+2}, \texttt{false})$ by some $E'_{r+2} \in \mathcal{W}'_{r+2}$ that strongly sees a supermajority of `false`-votes and has the ability to vote before E_{r+2}. Finally, $E'_r \notin \mathcal{F}'_r$.

Hence, $\mathcal{F}_r = \mathcal{F}'_r$ for all $r \in \mathbb{N}$. This concludes this lemma. ■

Corollary 4.
Let G, G' be the two hashgraphs as in Lemma 8, and let $\mathcal{U}_r, \mathcal{U}'_r$ denote set of all unique famous witnesses in round r for G and G', respectively. Then it holds true that $\mathcal{U}_r = \mathcal{U}'_r$ if the fame state for all witnesses in $\mathcal{W}_r$ is decided.

Proof. By Lemma 8 it holds $\mathcal{F}_r = \mathcal{F}'_r$ if the fame state for all witnesses in $\mathcal{W}_r$ is decided. Clearly, if $E \in \mathcal{U}_r$, it must also be $E \in \mathcal{U}'_r$ since there can be no new famous witness in $\mathcal{F}'_r$ with the same creator. Analogously, $\mathcal{U}'_r \subseteq \mathcal{U}_r$. ■

Corollary 5.
Let G, G' be the two hashgraphs as in Lemma 8, and let $r \in \mathbb{N}$ be an arbitrary round. If $\mathcal{W}_r \neq \emptyset$ and the fame state for all witnesses in $\mathcal{W}_r$ is decided, then it holds true that also the fame state for all witnesses in $\mathcal{W}'_r$ is decided.

Proof. We have $\mathcal{W}_r \subseteq \mathcal{W}'_r$ by Corollary 1 and $\mathcal{F}_r = \mathcal{F}'_r$ according to Lemma 8. This lemma also proved that non-famous witnesses are decided to be not famous in G' (cf. implication (2)). Moreover, the proof for $\mathcal{F}'_r \subseteq \mathcal{F}_r$ in Lemma 8 demonstrates for all witnesses $E'_r \in \mathcal{W}'_r \setminus \mathcal{W}_r$ the existence of a round $r + 2$ witness that decides E'_r to be not famous. Therefore, the famousness for all witnesses in $\mathcal{W}'_r$ is decided. ■

Lemma 9 (The Hashgraph Algorithm Achieves Self-Consistency).
Let G, G' be two hashgraphs such that $G \subseteq G'$ and both satisfy basic graph correctness. Further, suppose less than $\frac{1}{3}n$ forking nodes in G'. We denote with $\mathcal{H}_{alg}$ the hashgraph algorithm (i.e., `divideRounds`, `decideFame`, `findOrder`*) that some honest participant locally runs with some (local) hashgraph as input.*

Then, the following statement holds true for each event $E \in G$: If $\mathcal{H}_{alg}$ with input G assigned E a consensus position in the total order of events, then $\mathcal{H}_{alg}$ with input G' will assign E the same consensus position.

Proof. First, we note that the Hashgraph algorithm is deterministic; in particular for the function `decideFame`, the occurrence of a coin round, $i \mod \mathsf{coinRound}$ (cf. Figure 3.6), is predefined by the variable coinRound which is equal for all nodes. Further, voting events may "flip a coin" in a coin round by voting w.r.t. their signature; clearly, this is not nondeterministic: If G, G' share an event, they also share the same signature of this event. Therefore, $\mathcal{H}_{alg}$ running with the same hashgraph as input[5] will always output the same total order of events.

Let G, G' be the two hashgraphs from above, and assume $E \in G$ is an arbitrary event that $\mathcal{H}_{alg}$, with input G, assigned a consensus position x. That is, `findOrder` assigned E a round received number $r_r \in \mathbb{N}$. Recall that, generally, an event $\widehat{E} \in \widehat{G}$ has round received number $\widehat{r}_r$, if the following three conditions are fulfilled in $\widehat{G}$:

(1) The fame for all witnesses in $\widehat{\mathcal{W}_r}$, for all $r \leq \widehat{r}_r$, is decided.
(2) $\widehat{E}$ is an ancestor of all unique famous witnesses in $\widehat{\mathcal{U}}_{\widehat{r}_r}$.
(3) $\widehat{r}_r$ is minimal, i.e., there is no $r < \widehat{r}_r$ fulfilling the two conditions above.

Because of $\mathsf{roundReceived}_G(E) = r_r$, all three above conditions are fulfilled for E in G.

We denote with x' the consensus position that $\mathcal{H}_{alg}$ with input G' assigns E. For this theorem, we must prove $x = x'$. Next, we define the *round received set* of r in G,

$$\mathcal{C}_r = \{E_0 \in G \mid \mathsf{roundReceived}_G(E_0) = r\} \subseteq G, \tag{5.4}$$

to bet the set of all events in G with round received r. The Hashgraph algorithm sorts events (as one can observe in the `findOrder` algorithm, see Figure 3.6) first by the round received number, then ties by a consensus timestamp, and any remaining ties lexicographically by the signature of events. Further, let Δ denote the relative consensus number of E in $\mathcal{C}_{r_r}$ (i.e., the position of E in $\mathcal{C}_{r_r}$ ordered by consensus timestamp, and then lexicographically by signatures), where $0 \leq \Delta \leq |\mathcal{C}_{r_r}|$. Analogously, let $\mathcal{C}'_r$ denote the round received set of r in G', and Δ' the relative consensus number of E in $\mathcal{C}'_{r_r}$. In order to prove this theorem, we show *(i)* $\mathcal{C}_r = \mathcal{C}'_r$ for all $r \leq r_r$ and *(ii)* $\Delta = \Delta'$; this implies

$$x = \left| \bigcup_{r=1}^{r_r - 1} \mathcal{C}_r \right| + \Delta \overset{(i)}{=} \left| \bigcup_{r=1}^{r_r - 1} \mathcal{C}'_r \right| + \Delta \overset{(ii)}{=} \left| \bigcup_{r=1}^{r_r - 1} \mathcal{C}'_r \right| + \Delta' = x'. \tag{5.5}$$

Also notice that by condition (3) the sets $\mathcal{C}_r$ (and $\mathcal{C}'_r$) are each pairwise disjoint for all $r \leq r_r$, i.e., $\mathcal{C}_{r_1} \cap \mathcal{C}_{r_2} = \emptyset$ (and $\mathcal{C}'_{r_1} \cap \mathcal{C}'_{r_2} = \emptyset$) for all $r_1, r_2 \leq r_r, r_1 \neq r_2$.

Proof of $\mathcal{C}_r = \mathcal{C}'_r$

We first show $\mathcal{C}_r \subseteq \mathcal{C}'_r$ and $\mathcal{C}_r \supseteq \mathcal{C}'_r$ subsequently.

[5] In our implementation, nodes may have different states for the internal variables isWitness, isFamous, and roundReceived. However, in Corollary 1 we proved that for two hashgraphs G, G', with $G \subseteq G'$, follows $\mathcal{W}_r \subseteq \mathcal{W}'_r$, and $\mathcal{F}_r \subseteq \mathcal{F}'_r$ (cf. proof of Lemma 8) for all rounds $r \in \mathbb{N}$. Moreover, we do not reset the variable roundReceived, because round received numbers do not change as we will demonstrate in this proof.

$\mathcal{C}_r \subseteq \mathcal{C}'_r$
Let $E_0 \in \mathcal{C}_r$ and $r \leq r_r$ be arbitrary; we first show $E_0 \in \mathcal{C}_r \Rightarrow E_0 \in \mathcal{C}'_r$, by proving $\mathsf{roundReceived}_{G'}(E_0) = r$, i.e., E_0 fulfills all three round received conditions for r in G'. First, because of $\mathsf{roundReceived}_G(E) = r_r$, we know by condition (1) from above that the fame for all witnesses in $\mathcal{W}_{r'}$, for all $r' \leq r_r$, is decided. By Corollary 5 is also the fame for all witnesses in $\mathcal{W}'_{r'}$, for all $r' \leq r_r$, decided; hence, E_0 satifies condition (1) in G'.

Furthermore, by Lemma 8 applies $\mathcal{F}_{r'} = \mathcal{F}'_{r'}$ for all $r' \leq r_r$. This together with $G \subseteq G'$ implies that E_0 is an ancestor of all (unique) famous witnesses in $\mathcal{F}'_r \subseteq G'$; hence, condition (2) is fulfilled by E_0 in G'.

Condition (3), i.e. r is minimal in G', still needs to be shown: We already know that the fame for all witnesses $\mathcal{W}'_{r'}$, for all $r' \leq r_r$, is decided in G'; therefore, we have to prove that E_0 is not an ancestor of all unique famous witnesses in $\mathcal{U}'_{r-1}$. Because of $E_0 \in \mathcal{C}_r$, E_0 cannot be an ancestor of all $\mathcal{U}_{r-1}$ witnesses, otherwise $\mathsf{roundReceived}_G(E_0) < r$, which is a contradiction to $E_0 \in \mathcal{C}_r$. By Corollary 4, $\mathcal{U}_{r-1} = \mathcal{U}'_{r-1}$; hence, E_0 is also not an anestor of all $\mathcal{U}'_{r-1}$. Therefore, condition (3) is satisfied by E_0 in G', and we have $\mathsf{roundReceived}_{G'}(E_0) = r$ and $E_0 \in \mathcal{C}'_r$.

$\mathcal{C}_r \supseteq \mathcal{C}'_r$
Conversely, we show the contrapositive, i.e., the implication $E_0 \notin \mathcal{C}_r \Rightarrow E_0 \notin \mathcal{C}'_r$. Let $E_0 \in G'$ be arbitrary but $E_0 \notin \mathcal{C}_r$. First, suppose there exists a $r' \leq r_r, r' \neq r$, such that $E_0 \in \mathcal{C}_{r'}$. But then, as we just proved above, immediately follows $E_0 \in \mathcal{C}'_{r'}$; thus $E_0 \notin \mathcal{C}'_r$. So, let us assume E_0 has either no round received in G or $\mathsf{roundReceived}_G(E_0) > r_r$, in what follows.

We want to show $\mathsf{roundReceived}_{G'}(E_0) \neq r$ to conclude $E_0 \notin \mathcal{C}'_r$. We already know that all witnesses in $\mathcal{W}_{r'}$, for all $r' \leq r_r$, are decided to be either famous or not; thus, for rounds $r' = 1, \ldots, r_r$, all famous and, in particular, unique famous witnesses $\mathcal{U}_{r'} = \mathcal{U}'_{r'}$ are known (cf. Lemma 8 and Corollary 4). Since, for all $r' \leq r_r$, $E_0 \notin \mathcal{C}_{r'}$ and round received condition (1) is fulfilled for E_0 in G, E_0 cannot be an anecstor of all $\mathcal{U}_1, \ldots, \mathcal{U}_{r_r}$. By $\mathcal{U}_1 = \mathcal{U}'_1, \ldots, \mathcal{U}_{r_r} = \mathcal{U}'_{r_r}$, we conclude E_0 has either no round received in G' or $\mathsf{roundReceived}_{G'}(E_0) > r_r$; hence, $E_0 \notin \mathcal{C}_r$.

Proof of $\boldsymbol{\Delta = \Delta'}$
It remains to show that all events in $\mathcal{C}_{r_r} = \mathcal{C}'_{r_r}$ are identically sorted by consensus timestamp, and remaining ties by their signature lexicographically. Clearly, H_{alg} will calculate the correct order for the latter if $\mathcal{C}_{r_r} = \mathcal{C}'_{r_r}$. So, it is sufficient to prove that events will receive the same consensus timestamp by `findOrder` with input G and G'.

The consensus timestamp of an event $E_0 \in \mathcal{C}_{r_r}$ is defined to be the median of the timestamps of the events in $\mathcal{S} \subseteq G$, which is defined as

$$
\begin{aligned}
\mathcal{S} = \{E_1 \in G \mid \exists E_2 \in G \colon\ & E_2 \in \mathcal{U}_{r_r} \\
& \wedge E_1 \textit{ is a self-ancestor of } E_2 \\
& \wedge E_0 \textit{ is an ancestor of } E_1. \\
& \wedge E_0 \textit{ is not an ancestor of } E_1\textit{'s self-parent}\}.
\end{aligned}
$$

Observe that each event $E_1 \in \mathcal{S}$ is a self-ancestor of a unique famous witness in $\mathcal{U}_{r_r}$. Clearly, with $\mathcal{U}_{r_r} = \mathcal{U}'_{r_r}$ and $G \subseteq G'$ immediately follows $E_1 \in \mathcal{S}'$; thus, $\mathcal{S} \subseteq \mathcal{S}'$. Conversely, we also have $\mathcal{S}' \subseteq \mathcal{S}$; thus all events in $\mathcal{C}_{r_r}, \mathcal{C}'_{r_r}$ are assigned identical consensus timestamps in G and G'. We conclude $\Delta = \Delta'$.

We have proven $\mathcal{C}_r = \mathcal{C}'_r$, for all $r \leq r_r$, and $\Delta = \Delta'$; by Equation (5.5) we conclude $x = x'$, i.e., $\mathcal{H}_{alg}$ assigns on input G event E the same consensus position in the total order of events as $\mathcal{H}_{alg}$ with input G'. ■

Theorem 2 (The Hashgraph Algorithm Achieves Consistency).
Let G, G' be two consistent hashgraphs, both satisfying basic graph correctness, such that $G \cup G'$ contains less than $\frac{1}{3}n$ forking nodes. Let $\mathcal{H}_{alg}$ denote the hashgraph algorithm (see Lemma 9).

Then, the following statement holds true for each event $E \in G$: If $\mathcal{H}_{alg}$ with input G assigned E consensus position x in the total order of events, then $\mathcal{H}_{alg}$ with input G' will assign E either no consensus position or x.

Proof. Let x be the consensus position of E by $\mathcal{H}_{alg}$ on input G. If $\mathcal{H}_{alg}$ with input G' assigns E no consensus position, the theorem is fulfilled. So, suppose $\mathcal{H}_{alg}$ with input G' assigns E consensus position x'. Moreover, let y be the consensus position E received by $\mathcal{H}_{alg}$ on input $G \cup G'$, where $G \cup G'$ inherits the basic graph corretness property from G and G'. By Lemma 9, we have $x = y$ and $x' = y$; thus, $x = x'$. This concludes this proof. ■

5.4 Accountability of Hashgraph w.r.t. Consistency

Definition 10 (Consistency of Hashgraph).
Consider a run ω as in Definition 3, where $\omega \notin \alpha$. Let nodes *be the set of Hashgraph node identities (as specified as a parameter for $\mathcal{P}^{\mathsf{H}}$), and assume J is an instance of $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$. Let $((pid_i, msglist_i), \sigma)$ be a state update received by J such that*

- $pid_i \in \mathsf{nodes}$,
- $\mathtt{verifySig}_{\mathsf{pk}(pid_i)}((pid_i, msglist), \sigma) = \mathtt{true}$, *and*
- $msglist_i \subseteq \mathbb{N} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^*$.

Further, we define $\mathcal{M}_i(\lfloor\omega\rfloor)$ to be the set of all $msglist_i$ (from pid_i) that have been reported up to point $\lfloor\omega\rfloor$.

Common Prefix. *We say two message lists $msglist_1, msglist_2 \in \mathbb{N} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^*$* share a prefix *if $msglist_1 \subseteq msglist_2$ or $msglist_2 \subseteq msglist_1$.*

Self-Consistency. *We call $pid_i \in$* nodes self-consistent at point $\lfloor\omega\rfloor$ *if all pairs of message lists $msglist_1, msglist_2 \in \mathcal{M}_i(\lfloor\omega\rfloor)$ share a prefix.*

Node-Consistency. *We say that $pid_1, pid_2 \in$* nodes*, $pid_1 \neq pid_2$,* are consistent at point $\lfloor\omega\rfloor$ *if all pairs of message lists $msglist_1 \in \mathcal{M}_1(\lfloor\omega\rfloor)$, $msglist_2 \in \mathcal{M}_2(\lfloor\omega\rfloor)$ share a prefix.*

The system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$ satisfies consistency in the run ω *if all nodes are self-consistent at every point of the run ω, and all pairs of nodes are consistent at every point $\lfloor\omega\rfloor$.*

Definition 11 (Accountability w.r.t. Consistency).
Consider a run ω as in Definition 3. Again, we denote with nodes *the set of all Hashgraph node identities and assume J to be an instance of $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$.*

γ_1 *(Self-Consistency).* *The security property γ_1 contains all runs, where $\omega \notin \alpha$ and there exists some participant $pid \in \mathsf{nodes}$ such that pid does not satisfy self-consisteny at some point $\lfloor \omega \rfloor$.*

γ_2 *(Node-Consistency).* *This security property contains all runs, where $\omega \notin \alpha$ and there exist two participants $pid_1, pid_2 \in \mathsf{nodes}$, $pid_1 \neq pid_2$, such that pid_1 and pid_2 are not consistent at some point $\lfloor \omega \rfloor$.*

We define the accountability constraint C_3^{H} with the property $\gamma = \gamma_1 \cup \gamma_2 \cup \alpha$ as

$$C_3^{\mathsf{H}} := (\gamma \Rightarrow \mathsf{dis}(pid_1) \mid \cdots \mid \mathsf{dis}(pid_n)), \tag{5.6}$$

and set the accountability property $\Phi_3 := \{C_3^{\mathsf{H}}\}$.

We say $\mathcal{P}^{\mathsf{H}}$ with parameters as in Definition 3 is individually accountable w.r.t. consistency if for all environments $\mathcal{E}$ and adversaries $\mathcal{A}$ the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$ is Φ_3-accountable w.r.t. J.

Lemma 10.
Let G_1, G_2 be two hashgraphs, both satisfying basic graph correctness, with $G_1 \cup G_2$ containing less than $\frac{1}{3}n$ forking nodes. We denote with $msglist_1$, $msglist_2$ the message list of ordered entries that $\mathcal{H}_{alg}$ outputs on input G_1 and G_2, respectively. Then, it holds true that $msglist_1$ and $msglist_2$ share a prefix, i.e., $msglist_1 \subseteq msglist_2$ or $msglist_2 \subseteq msglist_1$ (**).

Proof. Let G_1, G_2 be the hashgraphs from above. We denote with $E_1^1, E_2^1, \dots, E_{k_1}^1$ the total order of $k_1 \in \mathbb{N}$ events $\mathcal{H}_{alg}$ outputs with input G_1, i.e., the order of events pid_1 calculates locally by running the Hashgraph algorithm $\mathcal{H}_{alg}$. Analogously, let $E_1^2, E_2^2, \dots, E_{k_2}^2$ be the total order of $k_2 \in \mathbb{N}$ events by $\mathcal{H}_{alg}$ on input G_2. We know $G_1 \cup G_2$ has less than $\frac{1}{3}n$ forking nodes and both G_1 and G_2 satisfy basic graph correctness. Moreover, G_1, G_2 are consistent hashgraphs by Lemma 3. We therefore conclude with Theorem 2, $E_i^1 = E_i^2$ for all $i = 1, \dots, \min\{k_1, k_2\}$, i.e., pid_1 and pid_2 calculate the same prefix of ordered events.

It still needs to be shown that $msglist_1, msglist_2$ share a prefix. Observe in the implementation of `findOrder` in $\mathcal{P}^{\mathsf{H}}_{\texttt{node}}$ or $\mathcal{F}^{\mathsf{H}}_{\texttt{judge}}$ that `findOrder` calculates with input G_1 and G_2 the same order, say j, for an (yet) unordered message list entry $(\varepsilon, \mathit{txId}, \mathit{eventID}, \mathit{msg})$, where msg is from the transaction set txs of an ordered event $E_k^1 = E_k^2$, $k < \max\{k_1, k_2\}$. That is, `findOrder` adds $(j, \mathit{txId}, \mathit{eventID}, \mathit{msg})$ to the message list $msglist_i$ with input G_i, for $i \in \{1, 2\}$. This holds because of the common prefix of ordered events of G_1 and G_2 and the fact that `findOrder` iterates over all unordered entries, belonging to the same event, in the same order. Hence, $msglist_1$ and $msglist_2$ share a common prefix of ordered message list entries.

Notice that many of the above statements may not hold true with negligible probability. ■

Theorem 3 (Hashgraph Achieves Individual Accountability w.r.t. Consistency).
Consider runs for the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$ with parameters for $\mathcal{P}^{\mathsf{H}}$ as in Definition 3 and let J be an instance of $\mathcal{F}^{\mathsf{H}}_{\texttt{judge}}$. Then, it holds true that $\mathcal{P}^{\mathsf{H}}$ achieves individually accountability w.r.t. consistency.

Proof. Let $\mathcal{P}^{\mathsf{H}}$ be the Hashgraph protocol with parameters as described above, and let $\mathcal{E}$ be an arbitrary environment and $\mathcal{A}$ be an arbitrary adversary. Towards proving this theorem, we have to show that J ensures Φ_3-accountability for the system $\mathbb{Q} = \{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$. Again, we do this by proving fairness and completeness independently. As before, we consider only a single Hashgraph instance in this proof. We first state some observations and general procedures in $\mathcal{F}^{\mathsf{H}}_{\texttt{judge}}$:

Handling of state updates in $\mathcal{F}^{\mathsf{H}}_{\texttt{judge}}$ with invalid signatures

We discussed earlier for the proof of Lemma 2 that all (supposed) state updates $((pid, currentState), \sigma)$ submitted to J by some client are discarded if the containing signature is invalid with the tuple $(pid, currentState)$ (i.e., $\texttt{verifySig}_{\mathsf{pk}(pid)}((pid, currentState), \sigma)$ outputs $\texttt{false}$) since J cannot definitively decide if pid, the reporting client, or both misbehaved. Consequently, such invalid state updates are ignored and do not influence any processing in $\mathcal{F}^{\mathsf{H}}_{\texttt{judge}}$.

Handling of valid state updates w.r.t. basic correctness at $\mathcal{F}^{\mathsf{H}}_{\texttt{judge}}$

We first describe how J processes valid state updates (i.e., state updates containing a valid signature) and which circumstances can lead to some node pid being blamed:

- J first checks whether the message list $msglist$ of the state update is a subset of $\mathbb{N} \times \{0,1\}^\eta \times \{0,1\}^\eta \times \{0,1\}^*$; that is, J ensures condition *(ii)* of α_1.
- Next, J checks if condition *(i)* of α_4 is satisfied, i.e., all entries $(ctr, txId, eventID, msg) \in msglist$ have a valid transaction ID, namely $\texttt{hash}(msg) = txId$.
- Lastly, condition *(ii)* of α_4 is checked by J, i.e., $msglist$ from pid must be a consecutive sequence with no duplicates or gaps w.r.t. the parameter ctr; formally, there exists exactly one entry $(ctr, txId, eventID, msg) \in msglist$ for all $ctr = 1, \ldots, |msglist|$.

In Lemma 2 it was proven that J is fair and complete under the above three basic correctness conditions. Particularly, all the aforementioned conditions for basic correctness hold true in a run $\omega \in \gamma$, i.e., a run that violates consistency does satisfy basic correctness, by definition of γ. Since $\mathcal{P}^{\mathsf{H}}$ is accountable w.r.t. basic correctness, it remains solely to prove that J is fair and complete w.r.t. the security properties γ_1 and γ_2.

Handling of message list states of nodes

Analogous to clients, J stores in the local variable states for each node pid the longest known message list, i.e., $\mathsf{states}[pid] = msglist'$ expresses that $msglist'$ is the longest message list of pid, which was previously submitted by some client. After initialization of J applies $\mathsf{states}[pid] = \emptyset$, for all $pid \in \mathsf{nodes}$, since no client submitted any state update yet.

J receives a new longest message list from pid

If the received message list ($msglist$) contained in a valid state update passes all basic correctness conditions, and it is the longest known message list from pid so far, i.e., $\mathsf{states}[pid] \subsetneq msglist$, then J updates the state for pid by setting $\mathsf{states}[pid] \leftarrow msglist$. Notice that the older state of pid ($\mathsf{states}[pid]$ before the assignment of $msglist$) must be a subset of the newly received $msglist$. If this is not the case, then there is a violation of self-consistency and pid must have misbehaved (we discuss this case later).

Besides the internal variable states that stores all longest submitted message lists from each node so far, J manages another internal variable for storing the longest so far received message list from all nodes that have not been blamed for a violation of self-consistency (γ_1) or node-consistency (γ_2), namely $\mathsf{msglist}_{\mathsf{max}}$. This initially empty message list is set to $msglist$ if not only $\mathsf{states}[pid] \subsetneq msglist$ applies, but also $\mathsf{msglist}_{\mathsf{max}} \subsetneq msglist$ holds true. Observe that even if some node, say pid, violates self-consistency, it still holds true that $\mathsf{states}[pid] \subseteq \mathsf{msglist}_{\mathsf{max}}$. After J outputs a verdict for pid, due to the violation of self-consistency, J recalculates $\mathsf{msglist}_{\mathsf{max}}$ by including only nodes that have not violated consistency so far.

Nodes that J has determined to have misbehaved w.r.t. consistency are captured in the set $\mathsf{consistencyVerdicts}$. Misbehaving nodes are added to this set after J outputs a verdict for such nodes. But it still remains to be shown that J actually blames nodes that violate consistency. We will prove later that J indeed blames nodes that violate consistency; thus, $\mathsf{consistencyVerdicts}$ captures exactly all nodes that violated γ.

This completes the processing of a common state update in $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$ where node-consistency is not violated.

Forking nodes

J keeps track of all nodes that violate fork-freeness by means of the set $\mathsf{forkingNodes} \subseteq \mathsf{nodes}$. Observe in Figure 3.13 that J does not only output a verdict for a forking node but also adds the misbehaving node to the set $\mathsf{forkingNodes}$. Since all honest nodes report their local hashgraph G to J before running $\mathcal{H}_{alg}$ with input G to calculate the order of entries in the message list, J will know if $\mathbb{E}_{\mathsf{W}}$, the union of all reported hashgraphs so far, contains at least $\frac{1}{3}n$ forking nodes. If this is the case, J no longer outputs any verdicts regarding γ_1, γ_2 for the rest of the run. This is vital to ensure that J does not blame any honest nodes that may calculate false states, due to the missing precondition of less than $\frac{1}{3}n$ forking nodes of Lemma 10.[6]

Notice that two honest nodes can each have a hashgraph with less than $\frac{1}{3}n$ forking nodes, but the union of both hashgraphs, say G, contains $\frac{1}{3}n$ forking nodes. Besides, these two nodes may even calculate message lists that do not have a common prefix. But this implies by Lemma 10 that G has more than $\frac{1}{3}n$ forking nodes.

Verdicts of J w.r.t. consistency γ

Observe in Figure 3.14 of $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$ that there is exactly one verdict for security properties γ_1 and γ_2, namely $\mathsf{dis}(pid_{send})$ and $\mathsf{dis}(pid)$, where $pid, pid_{send} \in \mathsf{nodes}$. Clearly, these verdicts achieve individual accountability for the accountabillity constraint C^{H}_3.

Fairness. Analogously to the proof for fork-freeness, it has to be shown that $\Pr[\mathbb{Q}(1^\eta) \mapsto \{(J : \psi) \mid \lfloor\omega\rfloor \not\models \psi\}]$ is negligible as a function in η, i.e., J renders a false verdict ψ only in a negligible amount of runs. In the proof of Lemma 2, we already showed that J is fair w.r.t. the verdicts of basic correctness (α); therefore it is left to show that J is also fair for the verdicts capturing properties γ_1 and γ_2. In the following, let $msglist$ be a message list of a valid state update of an honest node $pid \in \mathsf{nodes}$.

J is fair w.r.t. the verdict of γ_1

One can observe that J outputs a verdict for the violation of self-consistency only if $\mathsf{states}[pid]$ and $msglist$ do not share a common prefix, i.e., $\mathsf{states}[pid] \not\subseteq msglist$ and $msglist \not\subseteq \mathsf{states}[pid]$. Recall that $\mathsf{states}[pid]$ stores the longest of all previously submitted message lists that were signed by pid.

[6] The precondition of less than $\frac{1}{3}n$ forking nodes can be traced back to the strongly seeing lemma.

The proof for fairness of property α_2 illustrated (basic graph correctness) that honest nodes only submit hashgraphs to J that satisfy all conditions for basic graph correctness. In particular, the aforementioned proof also demonstrates that hashgraphs of honest nodes fulfill basic graph correctness at every point during the run ω, since honest nodes only merge other hashgraphs that fulfill basic graph correctness into their local one.

Let G_1 be the hashgraph pid used to calculate $\mathsf{states}[pid]$, and G_2 be the hashgraph pid used to calculate $msglist$. If pid is honest, it will have submitted G_1 and G_2 to J. The judge J will not render any verdicts if it detects at least $\frac{1}{3}n$ forking nodes; thus, J is always fair if it detects $\frac{1}{3}n$ forking nodes. Suppose this is not the case, then it holds in particular that $G_1 \cup G_2$ contains less than $\frac{1}{3}n$ forking nodes (because $\mathbb{E}_{\mathsf{W}}$ contains less than $\frac{1}{3}n$ forking nodes). Hence, msglist and $\mathsf{states}[pid]$ share a prefix by Lemma 10 (**), and thus J will never blame an honest node for a violation of γ_1 (**).

J is fair w.r.t. the verdict of γ_2

Observe in the implementation of $\mathcal{F}^{\mathsf{H}}_{\mathsf{judge}}$ that J blames nodes for a violation of γ_2 only if $\mathsf{msglist}_{\mathsf{max}}$ and $msglist$ do not share a prefix. If this is the case, J recalculates pid's latest state (msglist) on the latest submitted hashgraph, say G, by pid and checks whether $\mathsf{states}[pid]$ is a subset of the recalculated state of pid. Let G' be the hashgraph pid used to calculate $\mathsf{states}[pid]$, i.e., pid ran $\mathcal{H}_{alg}$ with input G' which outputted $\mathsf{states}[pid]$. Honest nodes always report new hashgraphs (i.e., a merged hashgraph of a sync that pid accepted) to J. Thus, $G' = G$ or $G' \subsetneq G$ if G is a more recent hashgraph of pid (cf. basic correctness α_3).

We know G contains less than $\frac{1}{3}n$ forking nodes (otherwise J would know about it and discard any new received message lists). Accordingly, we can conclude with Lemma 10 that msglist and $\mathsf{states}[pid]$ share a prefix (**). Because of $G' \subseteq G$, $\mathsf{states}[pid] \subseteq \mathsf{msglist}$ follows. Hence, pid will not be blamed by J (**).

Consequently, J is fair w.r.t. the verdicts for γ except for a negligible amount of runs. That is, $\Pr[\mathbb{Q}(1^\eta) \mapsto \{(J : \psi) \mid \lfloor\omega\rfloor \not\models \psi\}]$ is negligible as a function in η.

Completeness. For completeness, it must be shown that $\Pr[\mathbb{Q}(1^\eta) \mapsto \neg(J : \Phi_2)]$ is negligible as a function in η. That is, J ensures the accountability constraint C_3^{H} in all but a negligible number of runs of the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$. Specifically, for a run $\omega \in \gamma$, J must state a verdict $\bigwedge_{i \in I} \mathsf{dis}(pid_i)$ for $\emptyset \neq I \subset \{1, \ldots, n\}$ to ensure individually accountability of Φ_3. As discussed above, all verdicts for runs in γ_1 or γ_2 accomplish individually accountability, and for runs in α we already showed that J satisfies individual accountability w.r.t. basic correctness (see Lemma 2). Therefore, it is left to prove that J is also complete for runs in $\gamma_1 \cup \gamma_2$. Subsequently, let ω be a run of γ_1, γ_2, or both (thus, basic correctness is always satisfied). For the rest of this proof, we differentiate whether *1.* $\omega \in \gamma_1$ or *2.* $\omega \in \gamma_2$:[7]

1. Let $\omega \in \gamma_1$.

Then, by definition of γ_1, there exists some $pid \in \mathsf{nodes}$ that does not satisfy self-consistency at some point $\lfloor\omega\rfloor$, i.e., some client(s) submitted message lists $msglist_1, msglist_2$ to J in separate valid state updates, each containing a valid signature σ_i of $msglist_i$ that was sigend by pid, for

[7]For proving completeness, the case $\omega \in \gamma_1 \cap \gamma_2$ is not relevant because it is already included in case *1.* and *2.*.

$i \in \{1,2\}$. W.l.o.g. let $|msglist_1| < |msglist_2|$. Next, we do a case distinction whether *(i)* J stated some verdict for pid preceding point $\lfloor\omega\rfloor$ or J identified at least $\frac{1}{3}n$ forking nodes (i.e., $|\mathsf{forkingNodes}| \geq \frac{1}{3}n$), or *(ii)* pid has not been blamed by J until point $\lfloor\omega\rfloor$.

Case (i): $\mathsf{dis}(pid) \in \mathsf{verdicts} \vee |\mathsf{forkingNodes}| \geq \frac{1}{3}n$
First, if $|\mathsf{forkingNodes}| \geq \frac{1}{3}n$, then J identified and blamed at least $\frac{1}{3}n$ nodes for violation of fork-freeness; therefore, J ensures the constraint C_3^{H} and completeness is satisfied for this case.

Suppose less than $\frac{1}{3}n$ forking nodes. Observe in Figure 3.14 of $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$ that J discards all message lists for nodes that are already blamed. Since basic correctness is satisfied, pid must have violated fork-freeness or consistency at some point prior to $\lfloor\omega\rfloor$. Trivially, J already blamed pid; thus, J also ensures the constraint C_3^{H} in this case.

Case (ii): $\mathsf{dis}(pid) \notin \mathsf{verdicts}$
It must be shown that J states in this run the verdict $\mathsf{dis}(pid)$ for the violation of self-consistency by pid. We prove this statement by showing the following proposition:

We claim that pid satisfies self-consistency in every point $\lfloor\omega\rfloor$ before $msglist_2$ was received by J, i.e., $msglist_2$ is the first message list received by J such that self-consistency is violated.

Direct proof
Let $msglist'_2$ be the message list from pid that was received by J such that pid violates self-consistency the first time in run ω. Accordingly, there exists another $msglist'_1 \subseteq \mathsf{states}[pid]$, so that $msglist'_1$ and $msglist'_2$ do not share a prefix. Particularly, it holds true that $msglist'_1 \nsubseteq msglist'_2$ and $msglist'_2 \nsubseteq msglist'_1$. But then we also have $\mathsf{states}[pid] \nsubseteq msglist'_2$. Therefore, $|\mathsf{states}[pid]| < |msglist'_2|$, or $\mathsf{states}[pid]$ and $msglist'_2$ do not share a prefix. From both cases we conclude $msglist'_2 \nsubseteq \mathsf{states}[pid]$. Observe in Figure 3.14 that J states the verdict $\mathsf{dis}(pid)$ if, as in our case, $\mathsf{states}[pid]$ and $msglist'_2$ do not have a common prefix. However, since $\mathsf{dis}(pid) \notin \mathsf{verdicts}$, it must be that $msglist'_2 = msglist_2$. We conclude that pid satisfies self-consistency before J received $msglist_2$; this proves the statement.

The above proof demonstrates in particular that J states a verdict for pid. We conclude that J is complete w.r.t. self-consistency (**).

2. Let $\omega \in \gamma_2$.
By definition of γ_2, there exist two nodes pid_1, $pid_2 \in \mathsf{nodes}$, $pid_1 \neq pid_2$ such that pid_1 and pid_2 are not consistent at some point $\lfloor\omega\rfloor$, i.e., there exist two message lists

$$msglist_1 \in \mathcal{M}_1(\lfloor\omega\rfloor), msglist_2 \in \mathcal{M}_2(\lfloor\omega\rfloor)$$

from valid state updates that were reported to J by some client(s) such that $msglist_1$ and $msglist_2$ do not share a prefix at this point $\lfloor\omega\rfloor$. Specifically, for the two message lists applies $msglist_1 \nsubseteq msglist_2 \wedge msglist_2 \nsubseteq msglist_1$. We have to prove for completeness that J states at least one verdict, blaming pid_1 or pid_2.

W.l.o.g. let $msglist_2$ be the latter of the two message lists that J receives. We fix $\lfloor\omega\rfloor$ to be the point in ω where J just receives $msglist_2$ without doing any further processing. If we talk about a point in ω prior to $\lfloor\omega\rfloor$, we assume J has not yet received $msglist_2$. To begin with, we assume that $\lfloor\omega\rfloor$ is the first point in ω such that node-consistency is violated. We will later show that J outputs additional verdicts for further node-consistency violations.[8]

First, assume there are at least $\frac{1}{3}n$ forking nodes, i.e., $|\mathsf{forkingNodes}| \geq \frac{1}{3}n$. But then J must have already blamed at least $\frac{1}{3}n$ nodes due to violation of fork-freeness before $\lfloor\omega\rfloor$; thus, J fulfills completeness under this circumstance.

So, suppose $|\mathsf{forkingNodes}| < \frac{1}{3}n$. Since $\lfloor\omega\rfloor$ is the first point in ω where node-consistency is violated, J stated no verdict for pid_1 and pid_2 w.r.t. node-consistency, due to J's fairness w.r.t. node-consistency, until $\lfloor\omega\rfloor$. If pid_1 or pid_2 violates self-consistency at point $\lfloor\omega\rfloor$ (or before), J will (have) output(ted) a verdict for pid_1 or pid_2, as demonstrated in the completeness proof of γ_1; hence, completeness for γ_2 is again satisfied. Therefore, we assume for the rest of this proof that pid_1 and pid_2 satisfy self-consistency at point $\lfloor\omega\rfloor$ and conclude $pid_1, pid_2 \notin \mathsf{consistencyVerdicts}$ at $\lfloor\omega\rfloor$. At the beginning of this proof we discussed that $\mathsf{msglist_{max}}$ is the longest state of all nodes until there is a node-consistency violation. Particularly, at point $\lfloor\omega\rfloor$ holds true that $msglist_1 \subseteq \mathsf{states}[pid_1]$ and $\mathsf{states}[pid_i] \subseteq \mathsf{msglist_{max}}$, for $i \in \{1, 2\}$.

Processing of J upon receiving $msglist_2$

We describe the processing of J upon receiving $msglist_2$: Since $msglist_2$ is a proper superset of $\mathsf{states}[pid_2]$ (otherwise, $msglist_2 \subseteq \mathsf{states}[pid_2]$ implies that $msglist_2 \subseteq \mathsf{msglist_{max}}$ and thus $msglist_1, msglist_2$ share a prefix which is clearly a contradiction), J sets $\mathsf{states}[pid_2] \leftarrow msglist_2$. Next, we have to prove that $\mathsf{msglist_{max}}$ and $msglist_2$ do not share a prefix. This is important so that J can call the internal function `generateReport`, which tries to recalculate the state of all nodes and blames participants for which this is not possible.

Proof: $\mathsf{msglist_{max}} \not\subseteq msglist_2$

From above applies $msglist_1 \subseteq \mathsf{states}[pid_1] \subseteq \mathsf{msglist_{max}}$. Hence, $\mathsf{msglist_{max}} \not\subseteq msglist_2$ (otherwise $msglist_1 \subseteq \mathsf{msglist_{max}} \subseteq msglist_2$, which is a contradiction to $msglist_1 \not\subseteq msglist_2$).

Proof: $msglist_2 \not\subseteq \mathsf{msglist_{max}}$

Suppose $msglist_2 \subseteq \mathsf{msglist_{max}}$ for the purpose of contradiction. This and the valid statement $msglist_1 \subseteq \mathsf{msglist_{max}}$ implies that $msglist_1$ and $msglist_2$ share a prefix. Based on this contradiction, we conclude $msglist_2 \not\subseteq \mathsf{msglist_{max}}$.

Therefore, $\mathsf{msglist_{max}}$ and $msglist_2$ do not share a prefix. Consequently, J calls the internal function `generateReport` since $msglist_2 = \mathsf{states}[pid_2]$. Due to $msglist_1 \subseteq \mathsf{states}[pid_1], msglist_2 = \mathsf{states}[pid_2]$ and $msglist_1, msglist_2$ not sharing a prefix, it must be that $\mathsf{states}[pid_1]$ and $\mathsf{states}[pid_2]$ also cannot share a prefix. This will be important in the function `generateReport`:

***Proof: J outputs a verdict in* `generateReport`**

We prove that J outputs a verdict for pid_1 or pid_2 in `generateReport`. Let G_1, G_2 be the latest submitted hashgraphs of pid_1 and pid_2, respectively. J recalculates the message lists for pid_1 and pid_2 by running $\mathcal{H}_{alg}$ with input G_1 and G_2. Since $\omega \notin \alpha$, G_1 and G_2 must satisfy basic graph correctness. Furthermore, the proof of Lemma 3 demonstrates that G_1, G_2 are consistent

[8] We stress that the additional verdicts are not necessary in order to prove that J is complete w.r.t. node-consistency.

hashgraphs. Notice that because $|\mathsf{forkingNodes}| < \frac{1}{3}n$, it holds in particular true that $G_1 \cup G_2$ has less than $\frac{1}{3}n$ forking nodes. Let $\mathsf{msglist}_1, \mathsf{msglist}_2$ be the message lists J recalculates with input G_1 and G_2, respectively. By Lemma 10 we conclude that $\mathsf{msglist}_1, \mathsf{msglist}_2$ share a prefix (**), but we know $\mathsf{states}[pid_1], \mathsf{states}[pid_2]$ do not have a common prefix. Thus, $\mathsf{states}[pid_1] \nsubseteq \mathsf{msglist}_1$ or $\mathsf{states}[pid_2] \nsubseteq \mathsf{msglist}_2$. Hence, J outputs a verdict for pid_1 or pid_2, and we conclude J is complete w.r.t. node-consistency in all but a negligible amount of runs of $\mathbb{Q}$.

Remark

There are many possible outcomes of `generateReport`: pid_2 is not necessarily misbehaving, J may output a verdict for both participants, or J may even output additional verdicts for participants other than pid_1 and pid_2. But we stress that at least either pid_1 or pid_2 is blamed, as the above proof demonstrates. The function `generateReport` can blame arbitrary many participants: Instead of recalculating the states for pid_1 and pid_2 only, the functions compares all known states of participants to the recalculated state in `generateReport`. If their is any discrepancy, i.e., the known state is not a subset of the recalulated state, then J blames the corresponding participant. By Lemma 10, all recalculated states in `generateReport` share a prefix. In particular applies for all nodes pid that are not blamed in `generateReport`, $\mathsf{states}[pid] \subseteq \mathsf{msglist}$, where $\mathsf{msglist}$ is the recalculated state of pid. Thus, $\mathsf{msglist}_{\mathsf{max}}$ is the longest state of all nodes, not violating consistency, after `generateReport` finishes; these are all nodes that are not in the set $\mathsf{consistencyVerdicts}$. Next, we show that J outputs additional verdicts for node-consistency violations.

Additional verdicts

Let pid_1, pid_2 be two nodes that violate node-consistency at some point $\lfloor\omega\rfloor$ in the run ω. Again, let $\lfloor\omega\rfloor$ be the first point in ω where J received both message lists, not sharing a prefix, of pid_1 and pid_2 without doing any further processing. We again assume that $\mathsf{forkingNodes} < \frac{1}{3}n$ and pid_1, pid_2 have not been blamed by J for any consistency violation until $\lfloor\omega\rfloor$, i.e., $pid_1, pid_2 \notin \mathsf{consistencyVerdicts}$. The only difference to the proof above is that we now allow possible node-consistency violations of other nodes prior to $\lfloor\omega\rfloor$. With one exception, the proof is analogous: In the former proof, we easily observed that $\mathsf{msglist}_{\mathsf{max}}$ must be the longest state of all nodes since there was no previous node-consistency violation (see above); but this may not be true in this case. However, because of the implementation of `generateReport`, it still applies that $\mathsf{msglist}_{\mathsf{max}}$ is the longest state of all nodes that have not violated consistency at $\lfloor\omega\rfloor$. Hence, $\mathsf{states}[pid_1], \mathsf{states}[pid_2] \subseteq \mathsf{msglist}_{\mathsf{max}}$ is again fulfilled upon $\lfloor\omega\rfloor$. The remaining proof is completely analogous and J is guaranteed to blame pid_1 or pid_2.

Conclusion. We have succesfully proved that J ensures Φ_3-accountability for the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$ with parameters as in Definition 3, by showing that J satisfies both fairness and completeness except for a negligible amount of runs, i.e., $\Pr[\mathbb{Q}(1^\eta) \mapsto \{(J : \psi) \mid \lfloor\omega\rfloor \not\models \psi\}]$ and $\Pr[\mathbb{Q}(1^\eta) \mapsto \neg(J : \Phi_3)]$ are negligible as a function in η. This concludes the proof. ■

5.5 Closing Discussion

In this section we mainly discuss some strengths and weaknesses of our iUC model for hashgraph, specifically concerning consistency.

First, all our security results depend on an ideal certificate authority functionality $\mathcal{F}_{\mathtt{cert}}$. Similar to the work by Graf et al. in [10], it can be shown that all results carry over if $\mathcal{F}_{\mathtt{cert}}$ is replaced by its realization protocol, where signing is again done using an EUF-CMA secure signature scheme.

What J models in reality. In our model, (honest) nodes report their hashgraph to J, which then uses this information to render verdicts. Additionally, J collects auxiliary evidence, in from of states from nodes, to capture violations of consistency. In a real run of the hashgraph protocol, this corresponds to the situation where a client finds an inconsistency between the states of different nodes. That is, some nodes have a divergent order of messages. We capture this property in our definition for node-consistency.

Upon determining such an inconsistency, nodes would have to provide their hashgraph, which they have used to calculate their last state. These evidences are then given to other parties (e.g. other nodes or independent third parties [10]) to determine which party is responsible for the inconsistency; this is done by following the judging procedures as defined in our iUC model. Concretely, evidences (the hashgraphs and message lists) are first checked whether they fulfill basic correctness in order to prevent malicious participants to escape a verdict by providing malformed evidence. Secondly, since we do not restrict our model to less than $\frac{1}{3}n$ dishonest participants, it must be assured that the union of all submitted hashgraphs contains less than $\frac{1}{3}n$ forking nodes before making any judgements w.r.t. consistency. Only if the previous two conditions are fulfilled, the judging procedure for consistency is executed. However, we proved that in any case at least one node will always be determined to have misbehaved. Besides, if one of the nodes withholds the requested hashgraph, it is trivially guilty for disrupting the protocol [10].

Determining a consistency violation. In the judging procedure, recognition of consistency violations is exclusively determined by means of submitted message lists of clients (i.e., the states of nodes). No additional data, in form of hashgraphs, from nodes is required before a consistency violation is detected. This is indispensable in a real run of the protocol because it allows clients or nodes to determine a violation of consistency by following the judging procedure in $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$. The function call `generateReport` corresponds the situation where a consistency violation is detected, and nodes have to submit their hashgraphs as evidence; this situation was described in the previous paragraph.

Relaxing requirements for real-world employments. By the occurrence of a consistency violation, nodes have the burden of proof. That is, an honest node must put forward valid evidence in form of a hashgraph that can be used to recalculate its last ordering of messages and thus justify its state. If an honest node is unable to provide such a hashgraph, it is guilty and will accordingly be blamed in our model. In reality, this is a comparatively strict requirement: In a real implementation of the Hashgraph algorithm, nodes may not store the complete hashgraph and may only keep unordered events (i.e., events with no consensus position in the total order of events). However, nodes may agree out of band on the order of the first m messages (more specifically, the ordering of message list entries). Since message list entries also contain the eventID of the event in which the message was included, nodes may also agree on the order of the first k_m events. By this, nodes must agree on the total order of all messages included in these k_m events. Therefore, nodes can report pruned hashgraphs without these k_m events, for which consensus is already achieved. Besides some formal inconsistency problems (such as the genesis event or the correct message list entry order), all proved results should hold true for such pruned hashgraphs. In particular, the Hashgraph algorithm should output the same order of events, for which consensus is not yet reached, for all equally pruned hashgraphs of honest nodes. Herewith, it can be again checked, following the judging procedure in `generateReport`, if all disputed subsets of states can be recalculated with the pruned hashgraphs. Therefore, misbehaving participants can be again undisputedly identified.

6 Conclusion

In this work, we succesfully applied the accountability framework from Küsters et al. in [14] to demonstrate that the Hashgraph protocol fulfills accountability w.r.t. consistency, i.e., individual nodes of a hashgraph instance can be rightfully hold accountable for misbehavior. To achieve this, we constructed in Section 3.2 an iUC model of the Hashgraph protocol with the extension of the $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$ protocol. We presented in Section 5.4 a rigorous proof that individual participants, violating consistency, can be identified and blamed. Moreover, due to the fairness property of the accountability framework, we conclude that honest nodes, following the hashgraph protocol, will never be accused of misbehavior by our judging procedure. Our proof for accountability w.r.t. consistency relies on three cornerstones: *1)* the possibility to hold nodes accountable w.r.t. basic correctness which prevents participants (nodes and clients) to submit malformed evidence, *2)* the ability to identify forking nodes, and *3)* the consistency of the hashgraph algorithm, i.e., honest nodes with different hashgraphs will assign events (eventually) the same consensus position in the total order of events.

1) In Section 5.1, we demonstrated that the Hashgraph algorithm satisfies basic correctness w.r.t. consistency. This result enables us to enforce structural properties of submitted hashgraphs and message lists. Particularly, the basic graph correctness property of hashgraphs is an indispensable prerequisite for almost all proofs in our work.

2) Secondly, we proved in Section 5.2 that Hashgraph is accountable w.r.t. fork-freeness. Furthermore, our implementation of $\mathcal{F}^{\mathsf{H}}_{\mathtt{judge}}$ is able to output multiple verdicts for different nodes, violating fork-freeness. This is of exceptional importance because it nullifies the precondition of having more than $\frac{2}{3}n$ honest nodes. Therefore, we were able to prove accountability w.r.t. consistency of the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^{\mathsf{H}}\}$ without assuming a supermajority of honest nodes. We emphasize that our results for accountability w.r.t. consistency would not hold without the detection of forking nodes.

3) Lastly, we put forward in Section 5.3 a complete proof that illustrates the consistency of the Hashgraph algorithm. This result is captured by Theorem 2. By that, we proved that the Hashgraph algorithm will not assign two different consensus positions of an event for two hashgraphs, satisfying basic graph correctness with less than $\frac{1}{3}n$ forking nodes, as input. This result is essential to show that our judging procedure for consistency is both fair and complete. Our proof is considerably more detailed compared to the proof presented in [2] by Leemon Baird. Therefore, we believe our proof is also of interest independent to our work for accountability.

In Section 5.4 we illustrated, using the three results above, that Hashgraph does achieve accountability w.r.t. consistency. Moreover, we demonstrated in our approach that the judging procedure is able to output multiple verdicts, as long as less than $\frac{1}{3}n$ forking nodes are detected. Furthermore, we illustrated how our judging procedure could be employed for real-world applications of the

Hashgraph algorithm, and discussed potential problems for real-world usages. We also demonstrated, without proving it formally, how these problems can be mitigated if participants agree on a prefix of the list of ordered messages.

Bibliography

[1] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. “Hyperledger fabric: a distributed operating system for permissioned blockchains”. In: *Proceedings of the thirteenth EuroSys conference*. 2018, pp. 1–15 (cit. on p. 9).

[2] L. Baird. “The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance”. In: *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep* 34 (2016) (cit. on pp. 9–11, 25, 34, 50, 67).

[3] L. Baird, M. Harmon, P. Madsen. “Hedera: A public hashgraph network & governing council”. In: *White Paper* 1 (2019) (cit. on pp. 9, 10).

[4] L. Baird, A. Luykx. “The Hashgraph Protocol: Efficient Asynchronous BFT for High-Throughput Distributed Ledgers”. In: *2020 International Conference on Omni-layer Intelligent Systems (COINS)*. 2020, pp. 1–7. DOI: 10.1109/COINS49042.2020.9191430 (cit. on pp. 9, 10).

[5] R. G. Brown. “The corda platform: An introduction”. In: *Retrieved* 27 (2018), p. 2018 (cit. on p. 9).

[6] J. Camenisch, S. Krenn, R. Küsters, D. Rausch. “iUC: Flexible Universal Composability Made Simple”. In: *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part III*. Vol. 11923. Lecture Notes in Computer Science. Springer, 2019, pp. 191–221. URL: https://publ.sec.uni-stuttgart.de/camenischkrennkuestersrausch-asiacrypt-iuc-2019.pdf (cit. on pp. 17, 18, 21, 22).

[7] R. Canetti, A. Jain, A. Scafuro. *Practical UC security with a Global Random Oracle*. Cryptology ePrint Archive, Paper 2014/908. https://eprint.iacr.org/2014/908. 2014. URL: https://eprint.iacr.org/2014/908 (cit. on p. 28).

[8] K. Crary. “Verifying the hashgraph consensus algorithm”. In: *arXiv preprint arXiv:2102.01167* (2021) (cit. on pp. 10, 11).

[9] Forbes. *Hedera Hashgraph Explored*. https://www.forbes.com/sites/forbesdigitalassets/2019/09/26/hedera-hashgraph-explored/#70dad16d7bd0. (Accessed on 2022-07-10). Sept. 2019 (cit. on p. 9).

[10] M. Graf, R. Küsters, D. Rausch. *Accountability in a Permissioned Blockchain: Formal Analysis of Hyperledger Fabric*. Tech. rep. 2020/386. Cryptology ePrint Archive, 2020. URL: https://publ.sec.uni-stuttgart.de/grafkuestersrausch-iacr-2020.pdf (cit. on pp. 10, 35, 40, 64).

[11] M. Graf, D. Rausch, V. Ronge, C. Egger, R. Küsters, D. Schröder. *A Security Framework for Distributed Ledgers*. Tech. rep. 2021/145. Cryptology ePrint Archive, 2021. URL: https://publ.sec.uni-stuttgart.de/grafrauschrongeeggerkuestersschroeder-iacr-2021.pdf (cit. on p. 27).

[12] L. Hedera Hashgraph. *Hedera Global Governing Council*. https://hedera.com/council. (Accessed on 2022-07-10) (cit. on p. 10).

[13] L. Hedera Hashgraph. *Hedera Governing Council Votes to Purchase Hashgraph IP, Commits to Open Source World's Most Advanced Distributed Ledger Technology*. https://hedera.com/blog/hedera-governing-council-votes-to-purchase-hashgraph-ip-commits-to-open-source-worlds-most-advanced-distributed-ledger-technology. (Accessed on 2022-07-10). Jan. 2022 (cit. on p. 9).

[14] R. Küsters, T. Truderung, A. Vogt. "Accountability: Definition and Relationship to Verifiability". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*. ACM Press, 2010, pp. 526–535. URL: https://publ.sec.uni-stuttgart.de/kuesterstruderungvogt-ccs-2010.pdf (cit. on pp. 10, 35, 36, 67).

[15] S. Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: *Decentralized Business Review* (2008), p. 21260 (cit. on p. 9).

[16] Swirlds, Inc. *GitHub - Swirlds Hashgraph Platform code for Open Review*. https://github.com/hashgraph/swirlds-open-review. (Accessed on 2022-07-10). 2016 (cit. on p. 9).

[17] The Hindu. *Can hashgraph succeed blockchain as the technology of choice for cryptocurrencies?* https://www.thehindu.com/sci-tech/technology/can-hashgraph-succeed-blockchain-as-the-technology-of-choice-for-cryptocurrencies/article23348176.ece. (Accessed on 2022-07-10). Mar. 2018 (cit. on p. 9).

[18] G. Wood et al. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32 (cit. on p. 9).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature