Beyond Blockchain Ballots: UC-Secure Layer-2 Voting and Governance

Raghav Bhaskar*
raghav.bhaskar@gmail.com

Pooya Farshim
Input Output, Switzerland
pooya.farshim@iohk.io

Matthias Fitzi
Input Output, Switzerland
matthias.fitzi@iohk.io

Aggelos Kiayias
University of Edinburgh &
Input Output, UK
aggelos.kiayias@ed.ac.uk

***Abstract*—Maintaining a decentralized system requires a collective governance mechanism that allows participants to agree on changes to the system. In particular, the governance mechanism should offer a voting functionality for casting, collecting, and tallying votes in a confidential yet verifiable manner. Scaling this functionality for millions of participants in a cost-effective manner is a critical requirement for permissionless blockchains that remains unmet.**

We put forward a "layer-2" approach to meet this requirement in a setting where a permissionless blockchain acts as the fallback "layer-1" mechanism. Specifically, our approach to scalability realizes the protocol in a layer-2 fashion: the bulk of the protocol is executed off-chain, but secured on the blockchain with a minimal footprint.

We prove our protocol secure in the Universal Composability (UC) framework. First, we formalize a governance ideal functionality $\mathcal{F}_{\mathsf{L2Gov}}$. Our definition offers high levels of confidentiality and verifiability. Moreover, in the case of misbehavior, it allows faults to be attributed so that appropriate action (such as the slashing of on-chain funds) can be taken. Second, we demonstrate that our protocol UC-realizes the $\mathcal{F}_{\mathsf{L2Gov}}$ functionality based on a blockchain, an off-chain bulletin board, a distributed homomorphic encryption functionality ($\mathcal{F}_{\mathsf{DHE}}$), and other standard hybrids.

To the best of our knowledge, this work presents the first layer-2 blockchain voting protocol with a rigorous security analysis. We also point out some challenges that arise when applying the UC framework to layer-2 protocols.

***Index Terms*—Blockchain Governance, Voting, Bulletin Board, Verifiability, Universal Composability.**

I. Introduction

Blockchain systems operate in the long term and thus need to evolve their technologies and policies over time. This ability is critical, e.g., to adapt protocols according to new developments in cryptographic research, strengthen systems against new forms of attacks such as those stemming from quantum computing, enhance smart-contract functionalities for better usability, adjust policies in line with new regulatory constraints, or decide on an allocation strategy for treasury funds.

Since (permissionless) blockchains lack a central authority to decide and act on the evolution of the system, such decisions must be made in a decentralized way through some voting mechanism in a continuous process, commonly referred to as blockchain governance [1], [2].

Clearly, the voting aspect of blockchain governance should rely on secure electronic voting. Different options exist for implementing such a voting protocol, e.g., [3]–[5], and a natural choice is to use the blockchain itself. Indeed, electronic voting protocols are typically designed on top of a public bulletin-board, and the blockchain, as a public ledger, offers a natural and direct way to implement this.

Any such voting approach needs to consider that communication via a blockchain is typically slow (because of settlement times), and that storing data (e.g., the votes) on the blockchain can be prohibitively costly (due to transaction fees). In many, if not all, deployed cases of blockchain decision-making, this has led to limiting the inclusiveness of the "electoral roll": for instance, in Bitcoin, miners vote by choosing the version of the client software to run and they signal to each other which version they use in the blocks they generate.[1] In Cardano, a system of delegate representatives (Drep) exists[2] with each Drep receiving stake delegations and voting publicly on behalf of the stakeholders who delegated to them. Other systems that support on-chain decision-making are Tezos[3] and Polkadot.[4] In all these cases, the number of parties that can engage is subject to the throughput limitation of the underlying ledger, and hence an important question is whether they can support participation at a global scale (e.g., millions of votes possibly submitted in a short time-frame).

The scalability limitation mentioned above is widely discussed in blockchain systems as it impacts not only governance but also their primary functionality of recording transactions. Tackling this problem has led to the popularity of "layer-2" protocols such as roll-ups [6], [7] where transaction execution happens off-chain with minimal footprint on-chain for optimistic execution paths. Only in case of severe misbehavior, which is typically discouraged through, e.g., slashing of funds, inconsistencies must be reconciled on the blockchain (which is dubbed the "layer-1" in this context).

A. Context and motivation

In this paper, we focus on voting for blockchain governance with the goal of minimizing the use of the blockchain (and thus the cost) while allowing for some trade-offs that seem acceptable in this context.

*Work done while at Input Output, India.

[1] See signaling semantics in BIP 9 https://bips.dev/9/
[2] See https://www.1694.io/en
[3] See https://docs.tezos.com/architecture/governance
[4] See https://polkadot.com/opengov/

Note that blockchain governance differs from other voting use cases in several aspects. First, the voting power is typically determined by stake in the system—instead of per-ID individual suffrage. Second, maximal privacy is often not required as the topics are impersonal, and in the case of DRep voting privacy may even be undesired for the sake of accountability. Indeed, to the best of our knowledge, in most (if not all) of the voting-based blockchain-governance systems of major ecosystems (Cardano, Cosmos, Dash, Polkadot, or Tezos), the votes are cast as plaintexts.

We find it important to explore voting protocols for blockchain governance that minimize cost by avoiding excessive use of the blockchain, possibly accepting trade-offs motivated by what seems acceptable in the current blockchain-governance landscape. Indeed, we achieve such a trade-off by satisfying somewhat weaker-than-standard verifiability and privacy notions. Whereas our trade-off in verifiability seems inherent as we rely on an untrusted (but low-cost) layer-2 vote-collection mechanism, our trade-off in privacy is by choice to keep the protocol (and its analysis) relatively simple.

B. Contributions

We put forward a scalable layer-2 approach for blockchain-governance voting that enables low-cost secure decision-making for decentralized systems. A protocol preview is given in Section II (and Fig. 1).

Its security relies on an underlying smart-contract-enabled blockchain, but minimizes its use. We prove the protocol secure in the Universal Composability (UC) framework [8].

The protocol involves the following entities: voters, a bulletin-board manager (BBM), and a vote-tallying committee (VC). We assume a static, active (Byzantine) adversary who may corrupt any subset of these entities, with different levels of security guaranteed depending on which entities the adversary chooses to corrupt.

In addition to a (synchronous) blockchain, we assume the following *network* components: a synchronous network of pairwise authenticated channels between each voter and the BBM and a synchronous authenticated broadcast network among the VC.

We provide formal definitions of security for governance in the form of an ideal functionality in the UC framework [8]. Our definition ensures high levels of privacy and verifiability, which we define in Section III-A and demonstrate in Section VII by "translation" from our ideal functionality. Privacy ensures vote contents remain private under an honest-majority voting committee. Verifiability ensures the correctness of the final tally as well as the detection of vote omissions. Verifiability holds even when faced with a dishonest committee. We UC-realize this functionality modularly using, among other hybrids, a (weak) bulletin board functionality, a novel distributed homomorphic encryption functionality, and a public blockchain. Our formulations of the latter two are new. As the (optimistic) footprint on the blockchain remains low, the solution can also be attractive for general (non-blockchain) use cases. More concretely, our contributions are as follows.

Our governance system relies on homomorphic properties of ElGamal encryption. Traditionally, the security of such primitives has been developed in a property-based way. Here we give a novel and composable idealization of *distributed* homomorphic encryption as a UC ideal functionality. It can encrypt, homomorphically evaluate over ciphertexts, and later decrypt the results in a *distributed* manner. Defining UC ideal functionalities for primitives that come with public homomorphic features is challenging. We partially overcome this by requiring proof-of-knowledge of plaintexts, which allows simulators to extract plaintexts underlying evaluated ciphertexts. We realize $\mathcal{F}_{\mathsf{DHE}}$ based on a recent UC-secure distributed key-generation protocol due to Katz [9].

In developing our governance solution, we propose a formal definition of an ideal contract-enabled blockchain functionality ($\mathcal{G}_{\mathsf{MC}}$) that, we believe, strikes a good balance between ease of use and a sufficiently realistic modeling of blockchain. We also clarify subtleties that arise when applying the UC framework to layer-2 protocols. In particular, we discuss how the verification interfaces of various hybrids (digital signatures, NIZKs, etc.) can be accessed both from the on-chain and off-chain components of a layer-2 protocol, while retaining a modular approach to design and analysis.

Following the code-based game-playing framework of Bellare and Rogaway [10], and many works that build upon it, we present functionalities and protocols using fairly precise pseudo-code. This is somewhat in contrast to the tradition of using a mix of English prose and code in the UC literature. Such descriptions, although perhaps intuitive, leave open ambiguities that can lead to attacks, which are especially prevalent in concurrent settings such as UC. This added rigor can also bring the UC treatments of our protocols a step closer to formal verification and testing through tools such as `EasyUC` [11] and `SSProve` [12].

PAPER STRUCTURE. In the next two sections we give an extensive overview of our protocol and discuss how it relates to existing works. In Sect. IV we give an overview of the UC framework, and the basic syntax and notation that we use. In Sect. V we present $\mathcal{F}_{\mathsf{DHE}}$ and our UC realization, π_{DHE}. Sect. VI presents our layer-2 governance solution, including the functionality definition and its off-chain and on-chain components. We conclude the paper in Sect. VII where we discuss how our ideal functionality ensures the ballot-privacy and E2E-verifiability properties, and the technical challenges in our UC security proof. All mentioned ideal functionalities and protocols are described in detail (including pseudocode) in the appendices.

ACKNOWLEDGMENTS. We thank Sandro Coretti for participating in the early stages of this work, and Christian Badertscher and Markulf Kohlweiss for discussions about the UC framework.

II. PROTOCOL PREVIEW

THE PROTOCOL IN A NUTSHELL. To scale to thousands of voters at a low cost, our solution implements a vote collection

protocol (where the votes are sent to a bulletin-board manager $\mathbf{BBM}$) and an off-chain vote tallying protocol among the vote-tallying committee ($\mathbf{VC}$). In addition, a series of checks and fallback mechanisms are implemented to detect misbehavior by any of the parties.

Consider Fig. 1 for a high-level view of our protocol. The top sequence of transitions represents the state of the smart contract on the main chain. Initially the $\mathbf{BBM}$ or the $\mathbf{VC}$ submits the contract on-chain (step 0). We assume the code of this contract is authentically available off-chain to all parties. The $\mathbf{BBM}$ initially posts a signature verification key vk on-chain (step 1). Then the $\mathbf{VC}$ (represented by green boxes) jointly computes a pk and posts it to the mainchain (step 2). Since the blockchain is not exclusively used as a bulletin board, votes can be censored in the final commitment. We address this by requiring that the BBM provide a *receipt* in the form of a digital signature σ when a voter (represented by a blue V box) submits their vote. If the voter does not get a receipt, they can instead directly vote on-chain (step 3).

After a predefined period, the BBM publishes a vector commitment (Merkle root) of the collected votes μ, an encrypted tally ET, and proofs of correct collection and evaluation π on-chain. The voters can now request proofs of inclusion ρ of their votes (c, σ) with respect to the commitment (step 4). If a proof is denied, voters can raise a complaint on-chain (step 5), which the BBM must resolve in time to avoid having its funds slashed (step 6). The contract waits at the dispute state for a while (step 7) so that voters have a chance to complain and the BBM to resolve them.

After the complaint phase, the $\mathbf{VC}$ engages in a distributed decryption of the (on-chain) encrypted tally, and publishes the result res, along with proofs of correctness π', to the blockchain (step 8). The tally is accepted only if the accompanying proof is correct and there are no unresolved complaints.

The protocol allows voters to cast their votes in a *private* and *verifiable* manner. Here, privacy means that the vote contents remain hidden as long as the majority of a voting committee (VC) tallying the votes remains honest [13]. Verifiability means that voters can check that their ballots are indeed included in a (finalized) bulletin board, operated by a bulletin-board manager (BBM). It also means that the final tally published by the VC correctly corresponds to the votes on the bulletin board, even under a misbehaving BBM and VC [14], [15]. Our protocol also offers two mechanisms to prevent censorship: either via verification of vote inclusion against the BBM, or by submitting the vote directly to the blockchain (at a higher cost) where the smart-contract logic prevents censorship.

The collection of votes, proofs of inclusion, computation of an encrypted tally, and its distributed decryption are the core components of our governance protocol. We modularize our treatment by introducing two ideal functionalities: Distributed Homomorphic Encryption ($\mathcal{F}_{\mathsf{DHE}}$) and Bulletin-Board ($\mathcal{F}_{\mathsf{BB}}$).

DISTRIBUTED HOMOMORPHIC ENCRYPTION (DHE). We present a DHE functionality ($\mathcal{F}_{\mathsf{DHE}}$), which idealizes a distributed protocol that lets users encrypt votes and later tally them privately. In this functionality, a Decryption Committee (DC) jointly creates a set of common parameters, including a public key. Users can then encrypt votes of their choice to obtain a verifiable ciphertext, i.e., encrypted votes can be checked for validity (e.g., that the cleartext is from a given range). Any list of votes can be 'aggregated' to obtain a single ciphertext, called an encrypted tally CT. Finally, the DC can jointly decrypt a given CT and obtain a tally result together with a proof of correct decryption which any party can verify.

Although arguably a natural notion, precisely defining an $\mathcal{F}_{\mathsf{DHE}}$ ideal functionality that is both realizable and offers adequate levels of security turns out to be somewhat intricate. For instance, $\mathcal{F}_{\mathsf{DHE}}$ should also guarantee correct decryption of 'aggregated' ciphertexts. We achieve this by requiring that the ideal-world adversary (the simulator) provide plaintexts for each of the ciphertexts underlying an aggregated ciphertext. In our realization, we attach proofs of knowledge of underlying plaintexts to encrypted votes, as well as proofs of knowledge of correct aggregation for aggregated ciphertexts. These, in turn, allow our simulator to output these plaintexts via the two knowledge extractors. When defining the joint decryption of a ciphertext, our functionality specifies the behavior of decryption when there is no agreement on the input encrypted tally. Finally, the verification interfaces of this functionality are defined to prevent the simulator from outputting non-verifying encrypted tallies. We require an honest-majority $\mathbf{DC}$ to achieve ballot privacy [13], but also require that verifiability holds for any number of corruptions. We realize $\mathcal{F}_{\mathsf{DHE}}$ using a recently proposed distributed key-generation functionality $\mathcal{F}_{\mathsf{KGen}}$ by Katz [9], standard hybrids, and ElGamal encryption. We refer to App. D for a detailed pseudo-code specification and further discussions. A compact, high-level description appears in Prot. 1.

BULLETIN BOARD. A second component of our main protocol is an off-chain bulletin-board functionality that allows the collection, receipt generation, and verification of vote inclusion.

We formalize this as an ideal functionality $\mathcal{F}_{\mathsf{BB}}$, which, in essence, combines the collection of votes with a digital signature functionality (for receipts) and vector commitments (for proof of inclusion). Earlier treatments of voting assumed a trusted bulletin board that ensured that all submitted votes were collected and that a single view of the bulletin board was presented to all voters. More recent works (e.g., [16]–[18]) look at the desirable properties of a bulletin board when a trusted party (implementing the bulletin board) is not available. In contrast, we define a *weak* functionality whereby the bulletin-board managers (BBMs) may present different sets of votes to different voters and where each of these may contain only a subset of all legitimate votes. This keeps the underlying trust assumptions and costs minimal, and its realizations (as a layer-2 protocol) simpler. (The rationale is that by posting a (short) commitment of the BB on-chain, voters can ensure a single consistent view of the BB.) We strengthen the functionality, though, so that it offers *receipts*

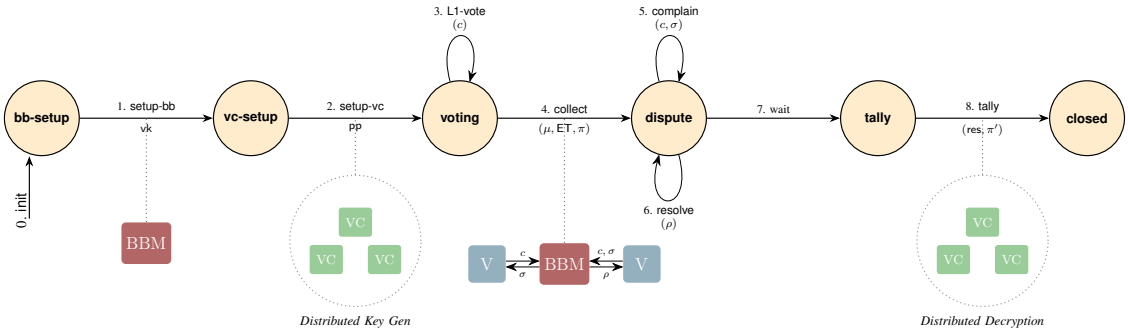

Figure 1. The high-level structure of our layer-2 governance protocol. See Section II for an overview.

of successful submissions as well as *proofs of inclusion*. These allow voters to detect censorship and to prove BBM misbehavior using the receipt if a vote is not included. Further discussion follows in the related-works section.

Our realization of $\mathcal{F}_{\mathsf{BB}}$ involves a *single* BBM and is simple: the $\mathbf{BBM}$ generates a signature key pair and publishes the verification key as pp. It then issues receipts as signatures on submitted data. The $\mathbf{BBM}$ provides proofs of inclusion via Merkle proof paths. Note that since there is a single BBM, there is no need for an agreement procedure on the submitted votes. Looking ahead, the fault attribution properties of our governance solution, coupled with layer-1 fallback (voting and complaints), ensure that a single BBM already offers strong forms of verifiability and censorship resistance. We still briefly discuss the added benefits of decentralized BBMs under future directions. We refer refer to App. E for a detailed specification and further discussion.

BLOCKCHAIN IDEALIZATION. Our protocol assumes the availability of a public blockchain, and thus requires an ideal functionality that specifies the operation and security guarantees of a blockchain. (The voting protocol can then be used for the governance of the blockchain itself.) Existing definitions of blockchains are either somewhat complex [19], which makes them difficult to employ as building blocks, or overly idealized and thus not realizable by actual blockchains [20]. Moreover, most definitions in the literature do not offer a validity function that supports a sufficiently general execution environment for smart contracts. We put forth a novel idealization of *smart-contract-enabled* blockchains that we believe strikes a good balance between ease of use and a sufficiently close modeling of concrete ledgers. Our formalization is reusable in other contexts, particularly where a contract system is required.

This functionality, which is modeled as a global subroutine and can be shared across multiple protocols, offers an interface to submit transactions and another to read the current state of the ledger. This abstraction offers liveness based on a *latency* parameter, which upper-bounds the time before which a valid transaction is included in the ledger, and a *slack* parameter, which specifies how close different parties' views of the ledger are. At the core of this definition lies a *transaction-validity* predicate that interprets transactions as contract code and executes them and determines validity with respect to the current contract state. This procedure allows us to specify smart contracts in a simple, yet precise manner.

GOVERNANCE. We now turn to layer-2 governance, $\mathcal{F}_{\mathsf{L2Gov}}$, a functionality that allows a set of voters to record and tally their votes in a *private* and *verifiable* manner. Furthermore, in case of certain types of adversarial attacks, $\mathcal{F}_{\mathsf{L2Gov}}$ allows users to take appropriate on-chain action to *circumvent censorship* or *raise complaints* about non-inclusion of votes. A vote-tallying committee $\mathbf{VC}$, a bulletin-board committee $\mathbf{BBM}$, and the voters $\mathbb{P}$ participate in $\mathcal{F}_{\mathsf{L2Gov}}$. The functionality comes with interfaces that, similarly to $\mathcal{F}_{\mathsf{DHE}}$, allow the creation of confidential (encrypted) votes and joint tallying of submitted votes. It also allows the collection of votes and generation and verification of proofs of inclusion similar to $\mathcal{F}_{\mathsf{BB}}$.

Concretely, $\mathcal{F}_{\mathsf{L2Gov}}$ allows voters to submit two types of votes, corresponding to layer-1 and layer-2 voting in our realization. In contrast to layer-2 votes, layer-1 votes are *guaranteed* to be included in the final set of votes even when faced with a dishonest-majority BBM. Layer-2 votes, on the other hand, are only included when the BBM is honest (though still with verifiability of vote inclusion). Why, then, not only offer layer-1 voting? The drawback is that layer-1 votes will be significantly more costly (and less efficient) to implement, in part due to transaction fees and slow settlement times associated with layer-1 actions. Thus, layer-1 votes are intended to be used only under "severe" adversarial scenarios.

$\mathcal{F}_{\mathsf{L2Gov}}$ also offers protection against malicious BBMs: it allows filing complaints against a BBM (which must issue receipts on collected ballots), and the voting result cannot be finalized if there are pending requests for vote-inclusion proofs that the BBM has not correctly resolved. Also, $\mathcal{F}_{\mathsf{L2Gov}}$ inherits the (universal) verifiability property of $\mathcal{F}_{\mathsf{DHE}}$ in that even a malicious $\mathbf{VC}$ cannot publish an incorrect tally result.

We note that $\mathcal{F}_{\mathsf{L2Gov}}$ is defined *without* any reference to $\mathcal{G}_{\mathsf{MC}}$. The idea is that $\mathcal{G}_{\mathsf{MC}}$ is a tool (hybrid) to realize $\mathcal{F}_{\mathsf{L2Gov}}$. This is in contrast to, say, roll-ups, which aggregate off-chain transactions. We further detail our abstraction in Section VI.

THE MAIN PROTOCOL. Our overview of $\mathcal{F}_{\mathsf{L2Gov}}$ already hinted at what a realization might look like. We now give further details of our realization π_{L2Gov}, highlighting in particular how

layer-1 and layer-2 votes are combined, as well as the contracts used.

π_{L2Gov} makes use of $\mathcal{F}_{\mathsf{DHE}}$, $\mathcal{F}_{\mathsf{BB}}$ (and thus, through universal composition, their realizations), the mainchain functionality $\mathcal{G}_{\mathsf{MC}}$, as well as a number of standard hybrids (digital signatures, NIZK, PKI, and secure/authenticated channels). The VC and BBM use the respective Setup interfaces of $\mathcal{F}_{\mathsf{DHE}}$ and $\mathcal{F}_{\mathsf{BB}}$ to generate their public parameters. They also generate signing keys and register them with a PKI. (PKI abstracts the specific way in which short-term (protocol-specific) keys and long-term on-chain keys are bound together.) The BBM then deploys a GovSC contract on the mainchain and initializes it with Merkle roots of the identities of the VC, BBM, and voters. The BBM signs and submits its public parameters to a GovSC contract, as does the VC. The contract accepts the VC parameters only if there are sufficiently many signatures.

To Vote, users first encrypt their votes via $\mathcal{F}_{\mathsf{DHE}}$ and then sign them. For layer-2 voting, they submit this data via $\mathcal{F}_{\mathsf{BB}}$, in response to which they obtain a receipt. For layer-1 voting, they submit the signed and encrypted vote to GovSC. As mentioned above, this layer-1 "fallback" feature is used when, for example, a dishonest BBM server refuses to issue a valid receipt. The contract will check the vote for correctness by verifying its signatures, PKI certificates, and validity proof, and add it to its internal layer-1 list of votes.

To finalize the list of votes, the BBM needs to ensure that layer-1 and layer-2 votes are appropriately combined. To this end, the BBM removes all layer-1 votes from its layer-2 votes that share the same user verification keys. It also removes all those with invalid proofs from layer-2 votes. (Looking ahead, when a complaint is submitted to GovSC, it will not be recorded unless it is also accompanied by a valid proof.) These (now disjoint) sets of votes are combined and their respective Merkle roots are computed. The BBM also prepares the encrypted tally CT using the homomorphic evaluation interface of $\mathcal{F}_{\mathsf{DHE}}$. It then attaches a succinct argument of knowledge (snark) proving that this encrypted tally corresponds to a set of votes having the computed Merkle root, and a distinct set of verification keys belonging to users, all of whom are eligible to vote. The proof also ensures that all ciphertexts are valid, correctly signed, and have valid PKI certificates. (We refer refer to App. F for the details of the circuit for which the snark is used.) The proof, Merkle roots, and CT are then posted to GovSC.

Verification of vote inclusion is implemented using $\mathcal{F}_{\mathsf{BB}}$. To file a complaint, voters submit their vote and its receipt to GovSC, which in turn checks them for validity and records them. To resolve a complaint, the BBM uses $\mathcal{F}_{\mathsf{BB}}$ to prepare an inclusion proof and submits it to GovSC. The tally and proof of its correctness (i.e., the result) are computed by the VC via the Decrypt interface of $\mathcal{F}_{\mathsf{DHE}}$. The result is signed and submitted to GovSC. The proof is checked on-chain, and if the proof is verified and there are no unresolved pending complaints, the tally is published as the election result.

In our formulation of $\mathcal{F}_{\mathsf{DHE}}$, we have made Eval an explicit (local) interface since the bulletin-board manager(s) (BBMs) will collect encrypted votes *and* compute an encrypted tally CT. CT, in contrast to the full bulletin board, is a short string and can be posted on the mainchain. This makes any faults due to either the BBMs not supplying the bulletin board to the VC or the VC not performing the decryption *attributable*. This allows appropriate action, e.g., slashing of locked funds, to be taken on-chain.

III. Related Works and Comparison

Some early research on electronic voting focused on defining the voter's privacy requirements, and on achieving verifiability of the voting process, resulting in formal definitions including ballot privacy [21], individual verifiability [22], universal verifiability [23] , and end-to-end verifiability [24].

End-to-End (E2E) verifiability has emerged as a desirable requirement, which informally means that the voter can verify that her ballot was "cast-as-intended", "recorded-as-cast", and "tallied-as-recorded". In [4], an E2E verifiable (and receipt-free) e-voting protocol is proposed. Their E2E security requires that an adversary cannot "deviate" the correct tally by more than a parameter d, which in turn depends on the number of voters that do not verify their votes and the size of the candidate set. In comparison, in our protocol, any honest voter can verify whether or not their ballot is included in the final ballots. Further, a zero-knowledge proof ensures that the output tally is indeed derived from the final ballots. Thus, in our protocol, in the worst case, the deviation in the result from the true tally depends on the number of unverified votes.

Several early works on electronic voting assumed a trusted bulletin board that is used to "publish" all votes. It was assumed to provide a consistent view to all and not to drop/miss any votes. In the absence of such a trusted bulletin board, a number of attacks have recently been shown. For instance, Hirschi et al. [25] identify a class of attacks where a malicious BB shows different bulletin boards to different voters such that individual and universal verifiability checks succeed, yet all voters agree on the same but incorrect result, i.e., E2E verifiability is broken. In our work, in contrast, we define a "weak" bulletin board functionality, where a malicious **BBM** may present different views of the BB to different voters and may also delete ballots. But in the final protocol we augment this with receipts, a requirement to commit a single finalized BB to a mainchain, and a complaint mechanism that, all together, allow a voter to detect any misbehaving **BBM**. Thus, our functionality can even be realized using a single untrusted server and still offer censorship resistance. We compare our work to blockchain voting in Section III-B.

A. Security properties

The focus of this work is to extend blockchain voting with a UC-secure layer-2 component to improve overall protocol efficiency. In particular, we do not aim for maximal security properties (such as coercion resistance), and restrict ourselves to a set of security guarantees that we deem sufficiently reasonable for blockchain governance. Concretely, our protocol satisfies the verifiability and privacy properties in Definitions 1

and 2. For space reasons, as game-based properties are not the focus of this paper, we only sketch the security properties rather than giving the full game-based definitions.

Before we discuss verifiability and privacy notions, we note that our protocol may abort without a result, either due to a corrupted BBM or a corrupted majority of the VC. While the above threshold condition for the VC is standard for committee-based protocols, requiring the single BBM to be honest may seem unreasonably strong (although standard for protocols involving a central (untrusted) authority). However, if the protocol aborts due to BBM misbehavior, the fault can be provably attributed to the BBM (in the sense of an MPC with identifiable abort [26]). Misbehavior of the (single) BBM can thus be disincentivized through economic mechanisms, such as "slashing" of bonded stake, as is common in systems like Ethereum.

VERIFIABILITY. The notion of *individual verifiability* ensures that each voter can verify whether or not its vote is included in the final tally. The notion of *universal verifiability* ensures that the tally correctly results from all votes cast to a public bulletin board. Whereas our protocol achieves individual verifiability, it achieves a weaker notion of universal verifiability because not all votes are publicly accessible. In particular, censored BBM votes may go undetected if the voter does not verify its vote for inclusion, i.e., eliminating the need for storing all votes on the blockchain comes at the cost of a weaker-than-standard notion of universal verifiability.

Still, the successful termination of our protocol implies that the BBM commits to a subset of the off-chain votes on the blockchain, and proves that the encrypted tally was correctly combined from all on-chain votes and the on-chain committed off-chain votes, and that all off-chain voters censored in this process finally abstained from verifying inclusion of their vote.

We capture this behavior along the lines of end-to-end (E2E) verifiability defined in [4]. In their protocol, all votes are written to a public bulletin board in a sub-protocol between voters and the election authority (EA). However, for an honest voter, this process can fail due to a corrupted EA, so that the vote gets censored. Their E2E verifiability notion takes this into account by requiring that the final tally account for exactly those honest votes that were successfully cast, while, from the public transcript (bulletin board), an "interpretation" of the adversarial votes can be output by a (not necessarily polynomial-time) *vote extractor* that makes the final tally correct (under the censorship of the unsuccessful voters).

As our protocol encourages the voters not to use the public bulletin board (i.e., the blockchain) but instead send their votes directly to the (untrusted) BBM, and moreover inclusion verification against the BBM is optional, we achieve a weaker notion of E2E verifiability than [4] as follows.

In the security game of [4], a challenger $\mathcal{C}$ takes the part of the honest voters. The adversary $\mathcal{A}$ controls the bulletin board, and, for each honest voter, instructs $\mathcal{C}$ to act on behalf of that voter to either cast a vote of $\mathcal{A}$'s choice or not engage in the protocol. $\mathcal{A}$ finally publishes the tally, with the winning condition being that the final tally cannot be "explained" by the vote extractor.

Our setup deviates in three aspects that require changes to their security game: voters have multiple options of where to cast their votes, votes not generating a BBM receipt may still end up being tallied (in [4], those votes do not), and verification of vote inclusion for BBM votes is optional (but mandatory in [4]). The main changes are: 1) votes cast (exclusively) to the BBM may or may not obtain a receipt, and 2) after the tally is committed (or published) on the blockchain, $\mathcal{A}$ may schedule inclusion-verification requests for the honest BBM-only voters, receiving the respective requests from $\mathcal{C}$.

Let there be n eligible voters, and let $\mathcal{D}$ be the domain of all possible vote choices, and let $\mathbf{v}_n \in (\mathcal{D} \cup \{\perp\})^n$ be a vector that represents the vote choices for all eligible voters where $\perp$ stands for not engaging in the protocol. Let H_{BC} be the set of indices of the honest voters who voted on the blockchain, and $H^{\mathrm{ver}}_{\mathrm{BBM}}$ and $H^{\mathrm{unver}}_{\mathrm{BBM}}$ be the sets of indices of honest voters who voted via BBM, and verified for vote inclusion, and did not, respectively. Finally, let a_{BC} be the number of (correct) adversarial votes cast to the blockchain (as can finally be inspected by $\mathcal{C}$, knowing how the honest voters behaved).

$\mathcal{A}$ wins the game if the protocol does not abort (i.e., yields a tally T, implying in particular that all honest vote-inclusion requests succeeded) and the extractor *fails* to output a vector $\tilde{\mathbf{v}_n}$ obtained from $\mathbf{v}_n$ by 1) leaving all elements with indices from $H_{\mathrm{BC}} \cup H^{\mathrm{ver}}_{\mathrm{BBM}}$ unchanged, 2) changing any number of elements with indices from $H^{\mathrm{unver}}_{\mathrm{BBM}}$ to $\perp$, and 3) assigning the elements with indices from adversarial voters arbitrarily subject to the restriction that a_{BC} elements must not be $\perp$ (as they represent valid blockchain votes), such that $T = \mathrm{tally}(\tilde{\mathbf{v}}_n)$.

Definition 1 (E2E Verifiability)**.** *A protocol achieves* E2E verifiability *if there is a vote extractor $\mathcal{E}$ such that every PPT adversary $\mathcal{A}$ wins the above security game with negligible probability.*

To conclude, our protocol achieves E2E verifiability as defined in Definition 1, which is weaker than universal verifiability because the cast ballots are not generally public. See also Fig. 2 for a comparison to other prominent blockchain-voting protocols with respect to verifiability.

PRIVACY. We achieve ballot privacy along the lines of the RPRIV property defined in [13], [27], which we briefly review. The (original) game involves two ballot boxes, BB_0 and BB_1, that are initially empty. The challenger $\mathcal{C}$ picks a random bit β. The adversary $\mathcal{A}$ gets access to BB_β and is allowed to either submit adversarial votes (in which case the vote is stored in both ballot boxes) or to instruct $\mathcal{C}$ to vote on behalf of honest voters by declaring an id and two vote inputs v_0, v_1, in which case vote v_0 is cast towards BB_0 and v_1 towards BB_1 under the given id. Finally, $\mathcal{C}$ computes the tally for each bulletin board. If the tallies are not equal, $\mathcal{A}$ loses. Otherwise, $\mathcal{C}$ opens the tally (more precisely, the full transcript the protocol revealed during the opening of the tally, including proofs etc.) to $\mathcal{A}$, and $\mathcal{A}$ must correctly guess β to win.

We also need to adapt this game for our purpose since we have two methods of casting votes (BBM and the blockchain BC), votes can get censored, and the optional verification requests for proof inclusion might leak additional information.

We thus apply the following changes to the game:

- In addition to the two ballot boxes BB_0 and BB_1 (representing the blockchain), there are also two BBMs, BBM_0 and BBM_1, where $\mathcal{A}$ is given access to the communication transcripts involving BBM_β.
- The adversarial vote requests can be directed either to the BB, BBM, or both. For BBM votes, the vote pairs (v_0, v_1) are also split, v_0 goes to BBM_0 and v_1 goes to BBM_1.
- After the vote-submission phase, for any selection of the previous BBM requests, $\mathcal{A}$ can demand that the effects be reverted (this models censorship). Only after this step, $\mathcal{C}$ computes the tallies for β and $1-\beta$. If the tallies differ, the adversary loses.
- $\mathcal{A}$ may schedule inclusion-verification requests (getting to see the communication between voter and BBM).

Definition 2 (Ballot (or Vote) Privacy)**.** *A protocol achieves* ballot privacy *if, for any PPT adversary $\mathcal{A}$, their advantage of guessing β is negligible.*

To conclude, our protocol achieves ballot privacy as defined in Definition 2. We note that we do not achieve voter anonymity, receipt-freeness, or coercion resistance. See Section III-B for a comparison to other prominent blockchain-voting protocols with respect to verifiability, and a brief comparison against the BPRIV ballot-privacy notion.

B. Comparison to other blockchain-voting protocols

We briefly compare our protocol to the most prominent blockchain-based voting protocols; see Fig. 2 for a summary.

The protocols in [28]–[30] are generic, while the protocols in [2], [31], [32] are tailored to blockchain governance.

Universal verifiability is achieved by all protocols in a trustless way, except for [32], which requires a trusted administrator, and our paper, which achieves E2E verifiability as defined in Definition 1 (still in a trustless way). We achieve this weaker notion because votes are encouraged to be submitted to the BBM but individual verification (of vote inclusion) is not mandatory; thus, censored votes by the BBM may go undetected. In contrast, all other protocols demand that all votes end up on the blockchain.

The protocols in [28]–[31] achieve the very strong notion of coercion resistance, the protocol in [32] achieves the weaker notion of voter anonymity (thus still a strong property), while our protocol and the one in [2] achieve ballot privacy, which is the weakest of the three notions. In particular, they achieve the RPRIV ballot-secrecy notion, which is weaker than the BPRIV notion developed in [13]. While, in contrast to verifiability, there seems to be no inherent trade-off that prevents us from achieving a strong privacy notion, we made this choice for simplicity (as our protocol and its analysis are already very complex).

Our work is perhaps closest to [2] in that it also treats UC-secure blockchain voting. We thus briefly extend our comparison to this paper. Their ideal functionality is closer to our governance functionality in that, unlike $\mathcal{F}_{\mathsf{DHE}}$, it does not operate at a primitive level but rather combines vote collection and tallying in a single functionality. It also does not offer a verification interface (as it relies on the availability of votes directly on a public blockchain). Moreover, their functionality directly uses the blockchain, and unlike our layer-2 solution, is not scalable. In particular, it does not have fallback or complaint interfaces, which are needed to retain high levels of security in our layer-2 solution.

Also, note that voting in the context of a proof-of-stake blockchain should weight the votes by the voters' stakes to faithfully represent the participants' power (and provide Sybil resistance). The protocol in [2] offers such stake-based features, though it sufficiently granularizes the stakes to render the computation of discrete logs (i.e., decryption) tractable. Similar techniques can also be applied in our protocol.

C. Future directions

The functionality and security of our scheme can be improved in a number of aspects: It might be desirable, e.g., to also decentralize the BBM (to reduce server load, or prevent a single dishonest BBM from forcing all voters to fall back on-chain). Our $\mathcal{F}_{\mathsf{BB}}$ already models more than one BBM, but we leave its UC realizations with many BBMs for future work. In blockchain settings, voters may want to delegate their votes, or it may be necessary to weight votes according to voters' stakes. These features require developing appropriate ideal functionalities to model them. Finally, we recall that we do not achieve the strongest possible privacy notion (as discussed above), and improving on this is an obvious next step. Also, we leave the treatment of economic incentives for future work.

IV. Preliminaries

THE UC FRAMEWORK. We formalize our security statements in the Universal Composability (UC) framework of Canetti [8]. The UC framework provides strong security guarantees in the sense that UC-secure protocols can be used securely and modularly in a wide variety of contexts. In short, in UC one compares the execution of a real protocol in the presence of an adversary to that of an ideal process in the presence of another adversary. If for any adversary against the real protocol there exists an ideal adversary (a.k.a. a simulator) against the ideal process so that no interactive distinguisher (a.k.a. the environment) can distinguish the two, the real protocol is said to UC-realize the ideal process.

Using this security definition, one can first design a (higher-level) protocol with respect to a simpler ideal process, and then replace the ideal process with a real process that UC-realizes it. This is essentially due to the UC composition theorem, the proof of which relies on the fact that the effects of any higher-level protocol can be absorbed into the environment.

	generic voting			governance voting			
	[28]	[29]	[30]	[2]	[31]	[32]	here
Verifiability	universal						E2E
corruption:	any					honest admin	any
Privacy:	coercion resistant			ballot privacy	coercion resistant	voter anonymity	ballot privacy
corruption:	any	honest maj.	1-out-2 honest admin	honest maj.	honest admin and maj.	any	honest maj.
Blockchain:	all votes						optional

Figure 2. Comparison of blockchain-voting protocols.

This replacement process can be extended to multiple ideal processes and enables a modular treatment of security for complex protocols. More details about the UC framework are given in App. A.

SYNTAX. We express our protocols and functionalities using a syntax of the form $\mathsf{P}.\mathsf{Interface}(\mathsf{input})$, defining what the party with PID P must do when run on input $(\mathsf{Interface}, \mathsf{input})$. This choice of notation leads to natural specifications of protocols and ideal functionalities as a collection of interfaces that, in the case of ideal functionalities, may share some state. Upon the first invocation of $\mathcal{F}$ with session ID sid, an instance of $\mathcal{F}$ is created with initial internal state specified in $\mathsf{Init}()$. In our ideal functionalities, a simulator (the ideal-world adversary) may be called. (This, however, is not the case for real protocols.) As a result, the execution is passed to the simulator, which in turn may hand the execution to a different functionality by calling one of its interfaces. We omit the SID of the caller and the callee from our specifications if the protocol is local and intended to be used by a single session of a calling protocol, or if the behavior of the interface is identical across different sessions. (The latter, for example, will be the case for shared verification interfaces.) The understanding is that different sessions are independent copies of the functionality that do not share any state. For shared (global) subroutines, however, we make the SID of the caller explicit to indicate how state is shared across different caller sessions. We assume only one session of shared subroutines. We assume every interactive machine has a read-only identity tape containing its identity (i.e., its party ID and session ID). Hence, we assume, w.l.o.g., that functionalities and protocols have access to their identity.

OVERVIEW OF SOME STANDARD HYBRIDS. We use a number of existing and new ideal functionalities. The core functionalities underlying our definitions are $\mathcal{F}_{\mathsf{DHE}}$, $\mathcal{F}_{\mathsf{BB}}$, and $\mathcal{G}_{\mathsf{MC}}$, which we discuss in the subsequent sections. Due to space limitations, we refer to the appendices for code-based specifications. To help readability, in the main body we only give informal, though more compact, overviews of these functionalities.

To formalize synchronous (but Δ-delayed) network functionalities, we need a mechanism that allows parties to proceed synchronously, round by round. Following standard approaches [33], we enable this via a shared *clock functionality* $\mathcal{G}_{\mathsf{Clock}}$ that keeps track of a global time/round counter shared across all machines registered with the clock. The functionality initializes the counter to zero, and increments it as soon as all registered machines express their willingness to proceed to the next round. This is done via the OK interface of the clock. A T interface provides awareness of the current round. The clock functionality is specified in App. B-H. All our protocols are formulated in this synchronous model.

$\mathcal{F}_{\mathsf{AC}}$ (App. B-A) idealizes a delayed authenticated channel (without confidentiality) where a party (with PID) P can send a message, using the interface Send, to a party Q, who can later fetch it using the interface Fetch on the sender PID. It guarantees that messages are authentic (i.e., indeed sent by the party whose PID is reported) and arrive with a delay of at most Δ rounds (as reported by the shared clock). $\mathcal{F}_{\mathsf{SC}}$ (App. B-B) idealizes delayed secure channels like delayed authenticated channels, except that $\mathcal{F}_{\mathsf{SC}}$ only leaks the length of messages to the simulator. $\mathcal{F}_{\mathsf{BC}}$ (App. B-C) idealizes a delayed authenticated broadcast channel. Users can broadcast a message using Cast and fetch a broadcast message using Fetch. It guarantees delivery of the same message to all parties, even if the sender is corrupt.

Functionality $\mathcal{F}_{\mathsf{CRS}}$ (App. B-D) idealizes access for all parties to a common reference string (with respect to a CRS generation algorithm D).

Func. $\mathcal{F}_{\mathsf{NIZK}}$ (App. B-E) idealizes a non-interactive zero-knowledge proof of knowledge with respect to a relation R. The simulator under the Prove interface generates proofs π for inputs $(x, w) \in R$. The simulator under Verify outputs a *witness*, which is used to decide the verification bit using R. The proofs are often assumed to be succinct (in which case we use the term snark), meaning that their size is small relative to the witness size. Func. $\mathcal{F}_{\mathsf{iNIZK}}$ idealizes an interactive zero-knowledge proof system. It is similar to the non-interactive functionality, but now, instead of verifying the correctness of the relation R locally on some input, the ideal functionality relies on another functionality, $\mathcal{F}^R$. Also, the identity $(\mathsf{pid}, \mathsf{sid})$ of the ITM running the ideal protocol corresponding to this functionality is specified in the statement.

Func. $\mathcal{F}_{\mathsf{Sig}}$ idealizes a digital signature scheme; it provides a key-generation Gen, a signature generation Sign, and a verification Verify interface. It guarantees that verification keys are distinct, correctly generated signatures verify, and verification rejects unless a signature has been correctly generated and verified with respect to the generated verification key, as long as the signer has not been corrupted (see App. B-F).

$\mathcal{F}_{\mathsf{PKI}}$ idealizes access to a PKI functionality where parties can register keys/data of their choice using $\mathsf{Register}$ and as a result obtain a certificate. Parties can verify certificates using $\mathsf{CertVerify}$ and also obtain the list of all registered identity-key pairs via $\mathsf{Retrieve}$.

Technically, we use *globally verifiable* variants of the above hybrids. These are variants whose verification interfaces accept calls from *any* session. To distinguish them from fully local

variants, we indicate globally verifiable functionalities with a gv superscript. We discuss this further in Section VII.

V. Distributed Homomorphic Encryption

We now present $\mathcal{F}_{\mathsf{DHE}}$, a novel functionality that, to the best of our knowledge, is the first to combine distributed key generation with homomorphic evaluation features. $\mathcal{F}_{\mathsf{DHE}}$ is used to realize the π_{L2Gov} protocol and influences the $\mathcal{F}_{\mathsf{L2Gov}}$ functionality. We then give a compact description of the protocol π_{DHE} (Prot. 1) realizing it. In App. D, we state $\mathcal{F}_{\mathsf{DHE}}$ in a synchronous model with respect to a common shared clock functionality (see Func. 13 in the appendix) that provides a consistent view of the current round to all parties that have registered with it. This choice, coupled with a synchronous network resource that delivers messages in at most, say, Δ rounds, allows us to define and realize desirable notions of *liveness*.

Concretely, $\mathcal{F}_{\mathsf{DHE}}$ comes with five interfaces: Setup, Encrypt, VerifyEnc, Decrypt, and VerifyDec (given in pseudocode in the appendix). Setup allows a Decryption Committee (DC) to generate parameters pp^*, and its structure follows the distributed key generation proposed by Katz [9]. The liveness of this interface ensures that as long as the DC is honest (i.e., at most t_{DC} of the DC members are corrupt), there is a common $\mathsf{pp}^* \neq \perp$ available to all honest DCs. Moreover, our definition ensures that there is agreement on pp^*, even when the DC is *dishonest*. (This is enforced by the shared global variable pp^*.)

The Encrypt interface models a local (non-interactive) procedure to prepare votes in a confidential manner as long as the DC is honest. The confidentiality guarantees stem from the conditions that 1) when the DC is honest, this interface reveals only the length of the message to the simulator, and 2) under decryption, only the final evaluation under F is revealed to the simulator. As a result, users obtain an encrypted and verifiable vote c.

Verifiability means that the ciphertext c can be checked, using the interface VerifyEnc, to contain a vote that respects a validity predicate Val. Ciphertexts implicitly embed proofs, and honestly generated ciphertexts are guaranteed to verify as valid, as VerEnc is populated under Encrypt.

The Eval interface allows the homomorphic aggregation (evaluation) of a list B of encrypted votes to obtain an encrypted tally CT. The Decrypt interface models the joint decryption of the ciphertext CT, producing a result res^* and a publicly verifiable proof of correctness of decryption π^*. (In defining this functionality, we assume that in realizations, a special error message is returned whenever a result-proof pair does not verify.) The functionality also provides liveness guarantees when the DC is honest: π^* is non-$\perp$ and must be a verifying proof. The decryption proofs π can be verified using a decryption interface VerifyDec.

As homomorphic evaluation is a public operation, it is somewhat unclear how to define an ideal functionality that guarantees the correctness of decryption over evaluated ciphertexts. Indeed, such homomorphic evaluation can be performed without using the interfaces of $\mathcal{F}_{\mathsf{DHE}}$, and still the Decrypt interface should recover the underlying plaintext correctly.

Decryption of homomorphically evaluated ciphertexts is the most complex aspect and the primary novel feature of this functionality.

We require that all encrypted votes as well as evaluated ciphertexts come essentially with implicit *proofs-of-knowledge* of correct evaluation. This, in turn, allows our simulator to output the plaintexts for all ciphertexts in an evaluated ciphertext via a two-step extraction procedure: first, the extractor recovers the list of (verifiable) ciphertexts underlying an encrypted tally; then, via a second layer of extraction, it obtains the votes underlying each of those ciphertexts.

In more detail, the Decrypt interface initially ensures that decryptors coordinate to decrypt an agreed-upon ciphertext CT^*. Following the above approach, the simulator is then required to extract the plaintexts underlying CT. In this context, the simulator may return a special plaintext value, $\mathrm{dec}(c)$, in which case the plaintext originally generated under Encrypt (and stored in the Dec list) is used. Finally, the result is computed via the evaluation function F. (Note that our functionality is explicitly defined to support a *single* function.) A proof π^* is supplied alongside the final decryption result res^*. The Decrypt interface ensures consensus on the pair (res^*, π^*), even in the presence of a dishonest majority.

The interface also populates Ver, ensuring that proofs subsequently verify via the VerifyDec interface. The latter also ensures that the result res is *unique* for each ciphertext. The validity of maliciously generated $(\mathsf{CT}, \mathsf{res})$ pairs remains at the discretion of the simulator: invalid $(\mathsf{CT}, \mathsf{res}, \pi)$ pairs may be generated when the VC is dishonest (and the generation of invalid proofs is always possible).

A final technical choice made in the definition of VerifyDec is whether the simulator should return a verification *bit* or a *witness*. Recall that $\mathcal{F}_{\mathsf{NIZK}}$ needs to decide if an input statement x is valid. This is defined by requiring the simulator to output a witness w and then using the relation $R(x, w)$ underlying the NIZK to decide the validity of x. In our context, a bit formulation is meaningful (and realizable) since the relations corresponding to the well-formedness of ciphertexts and decryption results are implemented through tables internal to the functionality. Thus, the relation can be efficiently decided by the simulator, without outputting a witness.

We realize $\mathcal{F}_{\mathsf{DHE}}$ in a $\mathcal{F}_{\mathsf{KGen}}$, $\mathcal{F}_{\mathsf{BC}}$, $\mathcal{F}_{\mathsf{NIZK}}$, and $\mathcal{F}_{\mathsf{iNIZK}}$ hybrid with ElGamal encryption in the exponent. $\mathcal{F}_{\mathsf{KGen}}$ is a distributed key-generation functionality, derived from a recent functionality due to Katz [9]. Briefly, the functionality in [9] is formulated for the specific case of prime-order groups and polynomial secret sharing and allows a set of n key-generation committee ($\mathbf{KC}$) members, among which at most t_{KG} are corrupted, to generate individual secret/public key shares $(\mathsf{sk}[i], \mathsf{pk}[i] = g^{\mathsf{sk}[i]})$ and agree on an overall public key pk. We extend this functionality to include the case where the number of corrupted parties in $\mathbf{KC}$ exceeds t_{KG}, thereby handling dishonest decryption committees. In this case, the polynomial used to derive the group key is chosen by the

simulator rather than the ideal functionality. It is easily verified that the UC-realization of $\mathcal{F}_{\mathsf{KGen}}$ via protocol Π_1 of Katz [9] based on simulation-sound (identity-based) NIZKs, a CRS functionality $\mathcal{F}_{\mathsf{CRS}}$, a PKI functionality $\mathcal{F}_{\mathsf{PKI}}$, and broadcast channels $\mathcal{F}_{\mathsf{BC}}$ also realizes our modification.

Prot. 1 gives a compact description of π_{DHE} (a full pseudocode version is given in App. D). $\mathcal{F}_{\mathsf{KGen}}$ is used to realize the Setup interface of $\mathcal{F}_{\mathsf{DHE}}$. Since the setup is group-based, we can implement the Encrypt interface of $\mathcal{F}_{\mathsf{DHE}}$ via ElGamal encryption augmented with a proof of knowledge of vote correctness (which VerifyEnc later verifies). The Eval interface exploits the homomorphic property of ElGamal encryption (for small votes). It checks the proofs for all input ciphertexts, aborting if any fails, and then computes the product of the ciphertexts, together with a proof of knowledge (via a NIZK hybrid) that 1) all input proofs verified and 2) the evaluated ciphertext was correctly multiplied. This proof needs to be succinct as the input set of ciphertexts can be large. The decryption interface Decrypt, run by the members of the DC, first verifies the evaluation proof π_{agg} from the encrypted tally $\mathsf{CT} = (\mathsf{CT}_1, \mathsf{CT}_2, \pi_{\mathsf{agg}})$. It then retrieves the secret key sk_{P} computed during Setup for party P and computes a decryption share $d_{\mathsf{P}} = (\mathsf{CT}_1)^{\mathsf{sk}_{\mathsf{P}}}$. Each party then attaches a proof π_{P}, using witness sk_{P}, of $(g, \mathsf{CT}_1, y_{\mathsf{P}}, d_{\mathsf{P}})$ being a valid DDH tuple (where g is the group generator). Each party broadcasts its decryption share and its proof to all other parties. After receiving the shares and proofs, the parties prune the proofs that do not verify. Finally, each party performs Lagrange interpolation (see also App. D) in the exponent at 0 to compute $\mathsf{D} = (\mathsf{CT}_1)^{\mathsf{sk}^*}$. This, in turn, can be used to decrypt the ElGamal ciphertext as usual. The result tally, along with the aggregation of all proofs proof, is returned as the final output. VerifyDec proceeds by verifying the aggregation proof, pruning the input proofs by removing all invalid ones, re-computing the decryption result (via interpolation in the exponent), and checking if it matches the claimed input result.

We state the security theorem for π_{DHE} informally below and give a formal theorem and proof in App. D. In Section VII, we discuss how this theorem can be interpreted in terms of a standard $\mathcal{F}_{\mathsf{NIZK}}$ functionality, which can be realized by concrete protocols for relations that place no calls to external protocols.

Theorem 1. *Protocol π_{DHE} as described in Prot. 1 UC-realizes $\mathcal{F}_{\mathsf{DHE}}$ with hybrids $\mathcal{F}^{\mathsf{gv}}_{\mathsf{NIZK}}$, $\mathcal{F}_{\mathsf{NIZK}}$, $\mathcal{F}_{\mathsf{iNIZK}}$, $\mathcal{F}_{\mathsf{KGen}}$, n sessions of $\mathcal{F}_{\mathsf{BC}}$ for $\mathsf{P} \in \mathbf{DC}$, and $\mathcal{G}_{\mathsf{Clock}}$.*

OTHER HYBRIDS. As mentioned in the introduction, our governance protocol also relies on a bulletin-board functionality $\mathcal{F}_{\mathsf{BB}}$ and our realization of it, π_{BB}. The complete pseudocode for the functionality and its protocol implementation are given in App. E under Func. 18 and Prot. 19. Our protocol also relies on a blockchain functionality, which we idealize as a functionality $\mathcal{G}_{\mathsf{MC}}$, which, while relatively simple, supports smart contracts and is a close model of a real blockchain. Existing idealizations are fine-tuned to real-world realization [19], making them somewhat cumbersome to use as hybrids, or else are too idealistic [20]. $\mathcal{G}_{\mathsf{MC}}$ is detailed in Func. 14 of App. C.

VI. LAYER-2 GOVERNANCE

We now describe the $\mathcal{F}_{\mathsf{L2Gov}}$ functionality that idealizes a layer-2 governance protocol. We will then describe how this functionality can be realized using a (possibly shared) mainchain $\mathcal{G}_{\mathsf{MC}}$ functionality, $\mathcal{F}_{\mathsf{DHE}}$, and $\mathcal{F}_{\mathsf{BB}}$, as introduced in the previous sections, as well as a number of standard hybrids.

The idealized governance functionality $\mathcal{F}_{\mathsf{L2Gov}}$ follows ideas in the definitions of $\mathcal{F}_{\mathsf{DHE}}$ and $\mathcal{F}_{\mathsf{BB}}$ that we presented earlier. An overview of $\mathcal{F}_{\mathsf{L2Gov}}$ is given in Func. 2. The complete pseudo-code is provided in Func. 20. The functionality shares some parameters with $\mathcal{F}_{\mathsf{DHE}}$ and $\mathcal{F}_{\mathsf{BB}}$, including t_{VC} and t_{BB}, the $\mathbf{VC}$ and $\mathbf{BBM}$ corruption thresholds, the vote validity predicate Val, and F, the tally computation function. Additionally, $\mathcal{F}_{\mathsf{L2Gov}}$ has a parameter $\mathsf{Val}^1_{\mathsf{BB}}$ that validates the set of layer-1 votes (e.g., by removing duplicates or invalid votes). The functionality also comes with a second layer-2 "bulletin-board" validity predicate $\mathsf{Val}^2_{\mathsf{BB}}$, which validates both layer-1 and layer-2 votes (e.g., deduplication *across* the two lists).

The BBSetup and VCSetup interfaces are defined similarly to the Setup interfaces in $\mathcal{F}_{\mathsf{BB}}$ and $\mathcal{F}_{\mathsf{DHE}}$, respectively. They ensure the liveness of parameters when their corresponding committees are honest. Agreement is ensured in all cases.

The Vote interface allows voters to cast votes in a private manner, either on "layer-1" or "layer-2". Layer-1 votes are always recorded (in V_1), and capture a perfect vote collection mechanism. In our realization, we use a blockchain to collect layer-1 votes. Layer-2 votes are only recorded if the BBM is honest (in V_2).

Our functionality satisfies the RPRIV notion from the introduction. Let $\mathcal{A}$ be an adversary against the RPRIV security of $\mathcal{F}_{\mathsf{L2Gov}}$. Consider a simulator that submits $\mathcal{A}$'s adversarial votes via corrupted parties and honest votes through the environment via honest parties. (This proceeds for both possible choices of $\beta \in \{0, 1\}$.) The sets of votes for each possible β must yield the same tally, since otherwise the adversary loses. Consequently, the same tally is revealed to the simulator under the Tally interface in both cases. As the Vote interface, under an honest $\mathbf{VC}$, withholds vote contents from the simulator, the votes remain hidden.

The FinalizeVotes interface is similar to the FormBB interface of $\mathcal{F}_{\mathsf{BB}}$ (which models both final BB formation and the servicing of the BB to any incoming requests), except that it now allows computing a finalized list V^* of collected votes from both layer-1 and layer-2. Liveness guarantees that the predicates $\mathsf{Val}^1_{\mathsf{BB}}$ and $\mathsf{Val}^2_{\mathsf{BB}}$ are satisfied.

The CheckInc interface combines the GetIncProof and VerifyIncProof interfaces of $\mathcal{F}_{\mathsf{BB}}$ into one. Thus, on receiving the validation bit b from the simulator, the functionality decides if b needs to be overwritten with 0 when the item associated with the submitted ciphertext is not in V^*. The Complain interface allows any party to file a complaint, potentially on behalf of others, by submitting a ballot. (This is

Protocol 1: π_{DHE}

$\underline{\mathsf{P.Setup}()}$ for $\mathsf{P} \in \mathbf{DC}, |\mathbf{DC}| = n$

$(\mathsf{pk}, [y_j : j \in [n]], \mathsf{sk}[\mathsf{P}]) \leftarrow \mathcal{F}_{\mathsf{KGen}}.\mathsf{P.Setup}()$
store $\mathsf{sk}[\mathsf{P}]$ and $\mathsf{pp}^* \leftarrow (\mathsf{pk}, [y_j : j \in [n]])$
return pp^*

$\underline{\mathsf{P.Encrypt}(\mathsf{pp}, m)}$ for $\mathsf{P} \in \mathbf{P} \cup \mathbf{DC}$

$(\mathsf{pk}, [y_j : j \in [n]]) \leftarrow \mathsf{parse}(\mathsf{pp})$
$r \leftarrow \mathbb{Z}_q; (c_1, c_2) \leftarrow (g^r, g^m \cdot pk^r)$
$\pi_{\mathsf{vote}} \leftarrow \mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathcal{R}_{\mathsf{VOTE}}].\mathsf{P.Prove}((\mathsf{pk}, c_1, c_2), (m, r))$
return $(c_1, c_2, \pi_{\mathsf{vote}})$

$\underline{\mathsf{P.VerifyEnc}(\mathsf{pp}, c)}$ for $\mathsf{P} \in \mathbf{P} \cup \mathbf{DC}$

$(\mathsf{pk}, [y_j : j \in [n]]) \leftarrow \mathsf{parse}(\mathsf{pp})$
$(c_1, c_2, \pi_{\mathsf{vote}}) \leftarrow \mathsf{parse}(c)$
$b \leftarrow \mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathcal{R}_{\mathsf{VOTE}}].\mathsf{P.Verify}((\mathsf{pk}, c_1, c_2)\ , \pi_{\mathsf{vote}})$
return b

$\underline{\mathsf{P.Eval}(\mathsf{pp}, \mathsf{B})}$ for $\mathsf{P} \in \mathbf{P} \cup \mathbf{DC}$

$(\mathsf{pk}, [y_j : j \in [n]]) \leftarrow \mathsf{parse}(\mathsf{pp})$
$\mathsf{CT}_1 \leftarrow \mathsf{CT}_2 \leftarrow 1_G$
for $(c_1, c_2, \pi) \in \mathsf{B}$ **do**
  if $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathcal{R}_{\mathsf{VOTE}}].\mathsf{P.Verify}((pk, c_1, c_2), \pi)$ **then**
    $(\mathsf{CT}_1, \mathsf{CT}_2) \leftarrow (\mathsf{CT}_1 \cdot c_1, \mathsf{CT}_2 \cdot c_2)$
  else
    return $\perp$
$\pi_{\mathsf{agg}} \leftarrow \mathcal{F}_{\mathsf{iNIZK}}[\mathcal{F}_{\mathcal{R}_{\mathsf{AGG}}}].\mathsf{P.Prove}((\mathsf{pk}, \mathsf{CT}_1, \mathsf{CT}_2), B)$
$\mathsf{CT} \leftarrow (\mathsf{CT}_1, \mathsf{CT}_2, \pi_{\mathsf{agg}})$
return CT

$\underline{\mathsf{P.Decrypt}(\mathsf{CT})}$ for $\mathsf{P} \in \mathbf{DC}$, once

$(\mathsf{pk}, [y_j : j \in [n]]) \leftarrow \mathsf{parse}(\mathsf{pp}^*)$
if $\mathsf{tally} \neq \perp$ **then return** tally
$(\mathsf{CT}_1, \mathsf{CT}_2, \pi_{\mathsf{agg}}) \leftarrow \mathsf{parse}(\mathsf{CT})$
if $\neg\mathcal{F}_{\mathsf{iNIZK}}[\mathcal{F}_{\mathcal{R}_{\mathsf{AGG}}}].\mathsf{P.Verify}((\mathsf{pk}, \mathsf{CT}_1, \mathsf{CT}_2), \pi_{\mathsf{agg}})$ **:return** $\perp$
$d_{\mathsf{P}} \leftarrow (\mathsf{CT}_1)^{\mathsf{sk}[\mathsf{P}]}$
$\pi_{\mathsf{P}} \leftarrow \mathcal{F}_{\mathsf{NIZK}}[\mathcal{R}_{\mathsf{DDH}}].\mathsf{P.Prove}((g, \mathsf{CT}_1, y_{\mathsf{P}}, d_{\mathsf{P}}), \mathsf{sk}[\mathsf{P}])$
$\mathcal{F}_{\mathsf{BC}}.\mathsf{P.Cast}(\mathsf{CT}, d_{\mathsf{P}}, \pi_{\mathsf{P}})$
$S \leftarrow \varnothing; \mathsf{Shares} \leftarrow [\,]$
for $j \in [n]$ **do**
  $(\mathsf{CT}^{(j)}, d_j, \pi_j) \leftarrow \mathcal{F}_{\mathsf{BC}}.\mathsf{P.Fetch}()$
  if $\mathcal{F}_{\mathsf{NIZK}}[\mathcal{R}_{\mathsf{DDH}}].\mathsf{P.Verify}((g, \mathsf{CT}_1^{(j)}, y_j, d_j), \pi_j)$ **then**
    $\mathsf{Shares} \leftarrow \mathsf{Shares} : d_j;\ S \leftarrow S \cup \{j\}$
    $\mathsf{proof}_j \leftarrow (d_j, \pi_j)$
  else $\mathsf{proof}_j \leftarrow \perp$
if $\mathsf{Maj}([\mathsf{CT}^{(j)} : j \in [n]]) \neq \mathsf{CT}$ **then return** $\perp$
$\mathsf{D} \leftarrow \mathsf{ExpLagrange}(t_{\mathsf{DC}}, S, \mathsf{Shares})(0)$
if $\mathsf{D} = \perp$ **then return** $\perp$
$\mathsf{tally} \leftarrow \mathsf{DL}(g, \mathsf{CT}_2/\mathsf{D})$
$\mathsf{proof} \leftarrow [\mathsf{proof}_j : j \in [n]]$
return $(\mathsf{tally}, \mathsf{proof})$

$\underline{\mathsf{P.VerifyDec}(\mathsf{pp}, \mathsf{CT}, \mathsf{tally}, \mathsf{proof})}$ for $\mathsf{P} \in \mathbf{P} \cup \mathbf{DC}$

$(\mathsf{pk}, [y_j : j \in [n]]) \leftarrow \mathsf{parse}(\mathsf{pp})$
$(\mathsf{CT}_1, \mathsf{CT}_2, \pi_{\mathsf{agg}}) \leftarrow \mathsf{parse}(\mathsf{CT})$
$[\mathsf{proof}_j : j \in [n]] \leftarrow \mathsf{parse}(\mathsf{proof})$
if $\neg\mathcal{F}_{\mathsf{iNIZK}}[\mathcal{F}_{\mathcal{R}_{\mathsf{AGG}}}].\mathsf{P.Verify}((\mathsf{pk}, \mathsf{CT}), \pi_{\mathsf{agg}})$ **then return** 0
$S \leftarrow \varnothing; \mathsf{Shares} \leftarrow [\,]$
for $j \in [n]$ **do**
  if $\mathsf{proof}_j \neq \perp$ **then** $(d_j, \pi_j) \leftarrow \mathsf{parse}(\mathsf{proof}_j)$
  if $\mathcal{F}_{\mathsf{NIZK}}.\mathsf{P.Verify}[\mathcal{R}_{\mathsf{DDH}}]((g, \mathsf{CT}_1, y_j, d_j), \pi_j)$ **then**
    $\mathsf{Shares} \leftarrow \mathsf{Shares} : d_j;\ S \leftarrow S \cup \{j\}$
$\mathsf{D} \leftarrow \mathsf{ExpLagrange}(t_{\mathsf{DC}}, S, \mathsf{Shares})(0)$
if $\mathsf{D} = \perp$ **then return** 0
return $(\mathsf{tally} = \mathsf{DL}(g, \mathsf{CT}_2/\mathsf{D}))$

Relations:

$\mathcal{R}_{\mathsf{VOTE}}((y, c_1, c_2), (m, r)) := \big(c_1 = g^r \wedge c_2 = g^m \cdot y^r \wedge \mathsf{Val}(m)\big)$
$\mathcal{R}_{\mathsf{AGG}}((y, C_1, C_2), B) := \big((C_1, C_2) = (\prod_{c \in B} c.c_1, \prod_{c \in B} c.c_2) \wedge \forall c \in B : \mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathcal{R}_{\mathsf{VOTE}}].\mathsf{P.Verify}((y, c.c_1, c.c_2), c.\pi_{\mathsf{vote}})\big)$
$\mathcal{R}_{\mathsf{DDH}}((g, h, g', h'), x) := \big(g' = g^x \wedge h' = h^x\big)$

used if checking the inclusion of votes through CheckInc fails.) The pending complaints are tracked in a list Com. The Resolve interface allows a BBM to resolve a pending complaint: the vote underlying the ballot is recovered, and the complaint is resolved and removed from Com only if it appears in V^*.

Individual verifiability of the functionality relies on CheckInc overwriting the simulator's output bit with 0 if a vote does not appear in V^*. This is a list maintained by the functionality that guarantees the inclusion of all layer-1 and layer-2 votes (as collected in the V_1 and V_2 lists). Note that this does *not* require the BBM (or the VC) to be honest, meaning that verifiability requires no trust assumptions beyond those needed for a secure blockchain.

The Tally interface idealizes the capability of the VC to form the tally tally^* for the votes on the finalized list. The tally is computed via a tally function F, which is a parameter of the system and operates over V^*. If the simulator-provided tally' differs from tally^*, $\perp$ is returned.

Recall that our notion of E2E verifiability essentially requires that the final tally can be "explained" in terms of honestly cast votes on the chain, verified layer-2 votes, some votes corresponding to unverified layer-2 votes, and adversarial votes. Observe that the functionality lets the simulator pick the final list of votes V^*. We claim that V^* is an explanation. The BBM is not trusted, and hence V^* is not sanitized. Still, V^* is always sanitized to contain all layer-1 votes; this subset of V^* corresponds to H_{BC} and a_{BC} adversarial indices. Now we consider other types of votes. 1) Verified honest layer-2 votes: Since CheckInc uses V^*, if passed, these votes must have been included in V^*. Thus, V^* contains the votes corresponding to $H_{\mathrm{BBM}}^{\mathrm{ver}}$. 2) Unverified honest layer-2 votes: Here the BBM may drop/censor some of these votes undetected. Consequently, V^* only corresponds to a subset of the indices in $H_{\mathrm{BBM}}^{\mathrm{unver}}$, instructing the challenger about which respective $\tilde{\mathbf{v}}_n$ entries to map to $\perp$. 3) Adversarial votes: The BBM can include arbitrary such votes in V^*, subject to the exclusion of the a_{BC}

Functionality 2: $\mathcal{F}_{\mathsf{L2Gov}}$ (compact)

P.BBSetup()
1: receive params $\mathsf{pp}_{\mathsf{bb}}$ from Sim
2: **if** $\mathsf{pp}^*_{\mathsf{bb}}$ not defined **then** $\mathsf{pp}^*_{\mathsf{bb}} \leftarrow \mathsf{pp}_{\mathsf{bb}}$
3: **if** $\mathbf{BBM}$ is honest **then** ensure $\mathsf{pp}^*_{\mathsf{bb}} \neq \bot$
4: **return** $\mathsf{pp}^*_{\mathsf{bb}}$

P.VCSetup()
5: receive params $\mathsf{pp}_{\mathsf{vc}}$ from Sim
6: **if** $\mathsf{pp}^*_{\mathsf{vc}}$ not defined **then** $\mathsf{pp}^*_{\mathsf{vc}} \leftarrow \mathsf{pp}_{\mathsf{vc}}$
7: **if** $\mathbf{VC}$ is honest ensure $\mathsf{pp}^*_{\mathsf{vc}} \neq \bot$
8: **return** $\mathsf{pp}^*_{\mathsf{vc}}$

P.Vote(layer, x)
9: ensure $(\mathsf{pp}^*_{\mathsf{bb}} \neq \bot) \wedge (\mathsf{pp}^*_{\mathsf{vc}} \neq \bot) \wedge \mathsf{Val}(x)$
10: **if** ($\mathbf{VC}$ is honest) **then then** receive ψ from Sim(layer)
11: **else** receive ψ from Sim(layer, x)
12: **if** (layer $= 1 \wedge \psi \neq \bot$) **then**
13: add (P, x) to V_1, if not already added
14: **if** (ψ, x) not already recorded as a valid cipher **then**
15: record it
16: **if** (layer $= 2 \wedge \psi \neq \bot \wedge \mathbf{BBM}$ is honest) **then**
17: add (P, x) to V_2, if not already added
18: **if** (ψ, x) not already recorded as a valid cipher **then**
19: record it and set $\mathsf{VerRec}[\psi] = 1$
20: **return** ψ

P.FinalizeVotes()
21: ensure $(\mathsf{pp}^*_{\mathsf{bb}} \neq \bot)$
22: **if** $\mathsf{T}() = T_4$ **then**
23: receive V' from Sim
24: **if** V^* not defined **then** $\mathsf{V}^* \leftarrow \mathsf{V}'$
25: ensure $\mathsf{Val}^1_{\mathsf{BB}}(\mathsf{V}^*, \mathsf{V}_1)$
26: **if** $\mathbf{BBM}$ is honest **then** ensure $\mathsf{Val}^2_{\mathsf{BB}}(\mathsf{V}^*, \mathsf{V}_1, \mathsf{V}_2)$
27: **return** ok
28: after round T_4 **return** whatever Sim returns

P.CheckInc(Q, ψ)
29: receive validation bit b from Sim
30: **if** (Q, ψ, b) is already recorded **then**
31: overwrite b with recorded bit
32: retrieve x using ψ from recorded messages (ψ, x)
33: **if** (x not retrieved or $(\mathsf{Q}, x) \notin \mathsf{V}^*$) **then**
34: overwrite b with 0
35: **if** ($\mathbf{BBM}$ is honest $\wedge$ $(\mathsf{Q}, x) \in \mathsf{V}^*$) **then**
36: overwrite b with 1
37: record (Q, ψ, b)
38: **return** b

P.Complain(Q, ψ)
39: receive ok from Sim
40: **if** $\mathsf{VerRec}[\psi] = 1$ **then**
41: add (Q, ψ) to list Com of complaints

P.Resolve(Q, ψ)
42: receive ok from Sim
43: retrieve x from recorded message (ψ, x)
44: **if** ($\mathbf{BBM}$ is honest $\wedge$ $(\mathsf{Q}, x) \in \mathsf{V}^*$) **then**
45: remove (Q, ψ) from list Com

P.Tally()
46: ensure $\mathsf{pp}^* \neq \bot \wedge \mathsf{V}^* \neq \bot$
47: $\mathsf{tally}^* \leftarrow F(\mathsf{V}^*)$
48: receive tally' from Sim(tally^*)
49: **if** tally not defined **then** $\mathsf{tally} \leftarrow \mathsf{tally}'$
50: **if** $(\mathsf{tally} \neq \mathsf{tally}^\star)$ **then return** $\bot$
51: **return** tally^*

P.GetResult()
52: receive res' from Sim
53: **if** res not defined **then** $\mathsf{res} \leftarrow \mathsf{res}'$
54: **if** $(\mathsf{Com} \neq \varnothing \vee \mathsf{res} \neq \mathsf{tally}^\star)$ **then return** $\bot$
55: **return** tally^*

blockchain votes. Since all such included votes are proven correct, there must be an assignment of the remaining elements in $\tilde{\mathbf{v}_n}$ such that the final tally is met.

We note that this functionality idealizes governance in a stand-alone way, and does not depend on the mainchain for its security *definition*. On the other hand, our realization uses the mainchain as we define it. An extended ideal functionality that models the monetary aspects of the protocol (e.g., slashing of funds) would also need to affect the mainchain where accounts are maintained. We leave this treatment (together with game-theoretic analysis) to future work.

We now describe Protocol π_{L2Gov} (see Prots. 3 and 4 for a compact description with more explicit versions deferred to App. F). This protocol uses distributed homomorphic encryption $\mathcal{F}_{\mathsf{DHE}}$, the bulletin board $\mathcal{F}_{\mathsf{BB}}$, the mainchain and clock functionalities $\mathcal{G}_{\mathsf{MC}}$ and $\mathcal{G}_{\mathsf{Clock}}$, a PKI functionality $\mathcal{F}_{\mathsf{PKI}}$, as well as $\mathcal{F}_{\mathsf{Sig}}$ and $\mathcal{F}_{\mathsf{NIZK}}$ as hybrids. In our realization, $\mathbf{VC}$ is a small- to medium-sized committee, while there is a single $\mathbf{BBM}$. Our protocol comprises an off-chain component (Prot. 3), whereby the VC, BBM, and voters interact without using the mainchain. It also has an on-chain component GovSC (Prot. 4), which is implemented via a contract deployed on the mainchain.

The VC and BBM use the respective Setup interfaces of $\mathcal{F}_{\mathsf{DHE}}$ and $\mathcal{F}_{\mathsf{BB}}$ to generate their public parameters. They also each generate a signing key and register it with a PKI. Specifically, functionality $\mathcal{F}_{\mathsf{PKI}}$ is used to bind the short-term signature verification keys of various parties to their identities (e.g., long-term on-chain keys). The use of $\mathcal{F}_{\mathsf{PKI}}$ abstracts away the specific ways this binding is achieved (e.g., through $\mathcal{G}_{\mathsf{MC}}$ itself). The BBM then deploys a GovSC contract on the mainchain alongside various initial data, such as Merkle roots of the identities of the VC, BBM, and voters. We assume that this data is common knowledge and is agreed upon out-of-band (e.g., using the blockchain itself). Thus, different parties can verify that they use the correct contract. We parameterize our functionality wrt. them.

The BBM signs and submits its public parameters to a GovSC contract, as does the VC. The contract accepts these if there are sufficiently many signatures on them, according to

two threshold values t_{BB} and t_{VC}.

To Vote, users first encrypt their votes via $\mathcal{F}_{\mathsf{DHE}}$ and then sign them. For layer-2 voting, they submit this data via $\mathcal{F}_{\mathsf{BB}}$, in response to which they obtain a receipt. For layer-1 voting, they submit the signed and encrypted vote via a mainchain transaction to GovSC. This is used, for example, when a dishonest BBM server refuses to provide a valid receipt. The contract checks each vote for correctness by verifying signature, PKI certificate, and the validity of the vote, and if it verifies, it stores it in memory.

An important component of our protocol is the finalization of the list of votes by the BBM under FinalizeVotes. In doing so, the BBM needs to ensure that layer-1 and layer-2 votes are appropriately combined. To this end, the BBM removes from the list of layer-2 votes all layer-1 votes whose user verification keys also appear on it. It also removes from layer-2 votes all those with invalid proofs. (Looking ahead, when a complaint is submitted to GovSC, it will not be recorded unless it is also accompanied by a valid receipt.) These (now disjoint) sets of votes are combined and their respective Merkle roots are computed. The BBM also prepares the encrypted tally CT using the Eval interface of $\mathcal{F}_{\mathsf{DHE}}$. It then attaches a succinct proof (SNARK) proving that this encrypted tally corresponds to a set of votes having the given Merkle root, for a distinct set of user verification keys, all of whom are eligible to vote. The proof also ensures that all ciphertexts are valid, correctly signed, and have valid PKI certificates. We refer to App. F for further details of the circuit for which the SNARK is computed. The proof, Merkle roots, and CT are then posted to GovSC. The actual Tally and the proof of its correctness are computed by the VC via the Decrypt interface of $\mathcal{F}_{\mathsf{DHE}}$. The result is signed and submitted to GovSC.

Verification of vote inclusion (CheckInc) is implemented by GetIncProof followed by VerifyIncProof (both of $\mathcal{F}_{\mathsf{BB}}$), and obtaining the necessary Merkle roots from (the memory of) GovSC. To Complain, a user submits their vote and a receipt to GovSC, which in turn checks them for validity and records them. To Resolve, the BBM uses GetIncProof of $\mathcal{F}_{\mathsf{BB}}$ to prepare an inclusion proof and submits it to GovSC.

The tally interface of GovSC checks if there are any pending complaints in the memory, and if not, finalizes the election result to that submitted in the Tally phase, if the tally proof verifies. Interface GetResult can be used to recover the election result from the memory of GovSC. As part of this, parties also check the correctness of the PKI parameters in the contract memory by comparing them to their local values.

We state the security theorem for π_{L2Gov} informally below; the formal theorem and the proof are provided in App. F-A.

Theorem 2. *Protocol π_{L2Gov} UC-realizes $\mathcal{F}_{\mathsf{L2Gov}}$ with hybrids $\mathcal{G}_{\mathsf{Clock}}$, $\mathcal{F}_{\mathsf{PKI}}$, $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}$, $\mathcal{G}_{\mathsf{MC}}$, $\mathcal{F}_{\mathsf{BB}}^{\mathsf{gv}}$, $\mathcal{F}_{\mathsf{iNIZK}}^{\mathsf{gv}}$, and $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}$.*

VII. UC Challenges

Our governance protocol consists of on-chain and off-chain components. For a fully modular design, we need to rely on components that are accessible from *both* of these parts. This approach, however, runs into a number of challenges.

First, the idealization of the blockchain needs to allow calls to other UC ideal functionalities. We enable this in our idealization of blockchains by interpreting calls to other (external) machines and forwarding them to the destination specified by the contract, and forwarding responses back.

The second challenge is that "dual access" to hybrids means that they are in essence *shared* (or global) as they offer (non-trivial) services to more than one protocol session. Shared functionalities are typically harder to formulate, realize, and use as hybrids. Similarly, hybrids other than the blockchain may also need to call shared subroutines (as is the case with $\mathcal{F}_{\mathsf{NIZK}}$), meaning that the callee, once again, is global and the caller not "subroutine respecting" in UC terminology, and hence cannot be modularly replaced.

Fortunately, however, the on-chain components of layer-2 systems (such as ours) only use the *verification* interfaces of cryptographic components (such as the verification interface of a zero-knowledge or a signature functionality), which typically use no secret data and are implemented in a public, stateless manner.

This observation leads to natural *globally verifiable* counterparts of the functionalities. Recall, e.g., that the signature functionality $\mathcal{F}_{\mathsf{Sig}}$ has a session ID sid^* that specifies the caller session it provides services to. (Typically, the session ID of the caller is required to match sid^*.) The globally verifiable variant of $\mathcal{F}_{\mathsf{Sig}}$ is defined similarly, except that its Ver interface accepts calls from *any* caller session, and then proceeds identically to this interface in $\mathcal{F}_{\mathsf{Sig}}$. The Gen and Sign interfaces of this functionality are left unchanged for a *single* session (in other words, calls from all but a single session are rejected).

One can realize $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}$ in the $\mathcal{F}_{\mathsf{Sig}}$-hybrid model essentially for *free* via a simple protocol ρ that runs $\mathcal{F}_{\mathsf{Sig}}$, and when it comes to Ver calls, on any caller sid, it calls $\mathcal{F}_{\mathsf{Sig}}$.P.Ver on sid^*, the single session ID that $\mathcal{F}_{\mathsf{Sig}}$ is designed for, and returns the result. (The SID of this functionality is defined so that it encodes sid^*.) We verify this claim below, but note that an immediate corollary, by the UC theorem, is that given any realization π of $\mathcal{F}_{\mathsf{Sig}}$, we get a realization of $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}$.

To prove the claim, consider a given (w.l.o.g. dummy) adversary $\mathcal{A}$ against $\rho^{\mathcal{F}_{\mathsf{Sig}}}$. We build a simulator $\mathsf{Sim}^{\mathsf{gv}}$ against $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}$ so that no environment $\mathcal{Z}$ can distinguish $\rho^{\mathcal{F}_{\mathsf{Sig}}}$ from $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}$. $\mathsf{Sim}^{\mathsf{gv}}$ runs a copy of $\mathcal{F}_{\mathsf{Sig}}$ internally and handles $\mathcal{Z}$'s queries (which are to ρ or $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}$) using it. The simulator uses its internal simulation of verification on sid^*, the SID of $\mathcal{F}_{\mathsf{Sig}}$ that it simulates, on any SID that $\mathcal{Z}$ uses when calling the Ver interface (of $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}$ or ρ). Note that $\mathcal{Z}$ cannot call the Gen and Sign interfaces as they reject on all SIDs $\neq \mathsf{sid}^*$, and the single sid^* is not external and cannot be used by $\mathcal{Z}$. This simulation is perfect as running $(\rho^{\mathcal{F}_{\mathsf{Sig}}}, \mathcal{A}, \mathcal{Z})$ is *syntactically* identical to running $(\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}, \mathsf{Sim}^{\mathsf{gv}}, \mathcal{Z})$: in the real world, ρ translates the SIDs, and in the ideal world, $\mathsf{Sim}^{\mathsf{gv}}$ does so.

Formulating globally verifiable variants of hybrids addresses the challenge of using verification interfaces from multiple sessions. However, a further subtlety arises when using $\mathcal{F}_{\mathsf{NIZK}}$.

Protocol 3: π_{L2Gov}

$\mathsf{S.BBSetup}()$: $\mathbf{BBM}$ participant S checks if the GovSC contract is already deployed, and deploys it if not. It creates a signing key vk and obtains a certificate cert on it and creates a $\mathsf{MerkleProof}$ π_{mr} for $\mathsf{S} \in \mathbf{BBM}$. It generates threshold key parameters $\mathsf{pp}_{\mathsf{bb}}$ using $\mathcal{F}_{\mathsf{BB}}.\mathsf{Setup}$, signs the message $(\mathsf{eID}, \text{“setup-bb”}, \mathsf{pp}_{\mathsf{bb}})$ using the corresponding secret key and submits it to $\mathcal{G}_{\mathsf{MC}}$.

$\mathsf{P.VCSetup}()$: $\mathbf{VC}$ participant P checks if the GovSC contract is already deployed, and deploys it if not. It creates a signing key vk and obtains a certificate cert on it and creates a $\mathsf{MerkleProof}$ π_{mr} for $\mathsf{P} \in \mathbf{VC}$. It generates threshold key parameters $\mathsf{pp}_{\mathsf{vc}}$ using $\mathcal{F}_{\mathsf{DHE}}.\mathsf{Setup}$, signs the message $(\mathsf{eID}, \text{“setup-vc”}, \mathsf{pp}_{\mathsf{vc}})$ using the corresponding secret key and submits it to $\mathcal{G}_{\mathsf{MC}}$.

$\mathsf{P.Vote}(\mathsf{layer}, x)$: Voter P generates a signing key vk and obtains a certificate cert on it and creates a $\mathsf{MerkleProof}$ π_{mr} for $\mathsf{P} \in \mathbf{P}$. It reads the GovSC smart contract's memory to obtain $\mathsf{pp}_{\mathsf{vc}}$. It encrypts its vote x under $\mathsf{pp}_{\mathsf{vc}}$, signs it using vk and then sends it to S (along with the cert) if submitting on layer 2; otherwise it submits it to $\mathcal{G}_{\mathsf{MC}}$. When submitting to the S, it waits to receive the receipt (c, ρ).

$\mathsf{S.FinalizeVotes}()$: S reads the GovSC smart contract's memory to obtain $\mathsf{pp}_{\mathsf{vc}}$ and BB_1. It computes the $\mathsf{MerkleRoot}$ μ_1 of BB_1, $\mathsf{MerkleRoot}$ v_{voter} of $\mathbf{P}$ and obtains (BB_2, μ_2) using $\mathcal{F}_{\mathsf{BB}}.\mathsf{FormBB}()$. It removes any invalid or duplicate votes from BB_2, and creates CT using $\mathcal{F}_{\mathsf{DHE}}.\mathsf{Eval}()$ and prepares a proof π_{bb} for relation $\mathcal{R}_{\mathsf{BBM}}$ with statement $(\mathsf{pp}^*, \mathsf{pp}_{\mathsf{vc}}, \mathsf{CT}, \mu_1, \mu_2, v_{\mathsf{voter}})$. It signs the statement and the proof under vk and submits it to $\mathcal{G}_{\mathsf{MC}}$.

$\mathsf{P.CheckInc}(\mathsf{Q}, \psi)$: Any participant P can check inclusion of the vote c (where $(c, \rho) \leftarrow \mathsf{parse}(\psi)$) in BB_1 or BB_2. To check in BB_1, it reads BB_1 from GovSC's memory and checks if $(\mathsf{Q}, \psi) \in \mathsf{BB}_1$. To check in BB_2, it can ask for an inclusion proof using $\mathcal{F}_{\mathsf{BB}}.\mathsf{GetIncProof}()$ and check the received proof using $\mathcal{F}_{\mathsf{BB}}.\mathsf{VerifyIncProof}()$.

$\mathsf{P.Complain}(\mathsf{Q}, \psi)$: Any participant P can file a non-inclusion complaint by submitting $(\mathsf{eID}, \text{“complain”}, \mathsf{Q}, \psi)$ to $\mathcal{G}_{\mathsf{MC}}$.

$\mathsf{S.Resolve}(\mathsf{Q}, \psi)$: The server S can resolve any complaint (Q, ψ) by computing an inclusion proof π using $\mathcal{F}_{\mathsf{BB}}.\mathsf{GetIncProof}(\mu_2, \mathsf{Q}, c)$ where $(c, \rho) \leftarrow \mathsf{parse}(\psi)$ and μ_2 is the commitment to BB_2 it submitted to $\mathcal{G}_{\mathsf{MC}}$. It signs the proof and submits it to $\mathcal{G}_{\mathsf{MC}}$.

$\mathsf{P.Tally}()$: Any $\mathsf{P} \in \mathbf{VC}$ can output a tally by participating in a joint decryption using $\mathcal{F}_{\mathsf{DHE}}.\mathsf{Decrypt}(\mathsf{CT})$, where $(\mathsf{CT}, \mu_1, \mu_2)$ are read from GovSC's memory. It creates a proof π_{mr} that $\mathsf{P} \in \mathbf{VC}$ and submits the resulting tally by signing it under vk and sending it to $\mathcal{G}_{\mathsf{MC}}$.

$\mathsf{P.GetResult}()$: Any participant P can obtain the result by checking PKI parameters and reading “fin-tally” from GovSC's memory.

Protocol 4: GovSC

$\mathsf{setup\text{-}bb}(\mathsf{P}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, \mathsf{pp}_{\mathsf{bb}}, \sigma)$: Contract checks if current state is $\mathsf{bb\text{-}setup}$ and time in $[T_0, T_1 - \Delta_1]$. If submitting party P is a $\mathbf{BBM}$ member and has a valid cert for vk, it stores $(\mathsf{vk}, \mathsf{pp}_{\mathsf{bb}})$ in memory, and if it finds more than t_{BB} entries for $\mathsf{pp}_{\mathsf{bb}}$ in memory, and “fin-pp-bb” not set, it adds (“fin-pp-bb”, $\mathsf{pp}_{\mathsf{bb}}$) to memory and moves to state $\mathsf{vc\text{-}setup}$.

$\mathsf{setup\text{-}vc}(\mathsf{P}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, \mathsf{pp}_{\mathsf{vc}}, \sigma)$: Contract checks if current state is $\mathsf{vc\text{-}setup}$ and time in $[T_0, T_1 - \Delta_1]$. If submitting party P is a $\mathbf{VC}$ member and has a valid cert for vk, it stores $(\mathsf{vk}, \mathsf{pp}_{\mathsf{vc}})$ in memory, and if it finds more than t_{VC} entries for $\mathsf{pp}_{\mathsf{vc}}$ in memory, and “fin-pp-vc” not set, it adds (“fin-pp-vc”, $\mathsf{pp}_{\mathsf{vc}}$) to memory and moves the state to voting.

$\mathsf{L1\text{-}vote}(\mathsf{P}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, c, \sigma)$: Contract checks if current state is voting and time in $[T_2, T_3 - \Delta_1]$. If submitting party P is a v_{voter} member and has a valid cert for vk, it checks P's signature on $(\mathsf{eID}, \text{“L1-vote”}, c)$, and if it finds no other entry for “L1-vote” in memory, it verifies correct formulation of c before adding (“L1-vote”, $\mathsf{P}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, c, \sigma$) to memory.

$\mathsf{collect}(\mathsf{S}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, \mathsf{CT}, \mu_1, \mu_2, v_{\mathsf{voter}}, \pi_{\mathsf{bb}}, \sigma)$: Contract checks if current state is voting and time in $[T_2, T_4 - \Delta_1]$. If submitting party P is a $\mathbf{BBM}$ member and has a valid cert for vk, it checks P's signature on $(\mathsf{eID}, \text{“collect”}, \mathsf{CT}, \mu_1, \mu_2, v_{\mathsf{voter}}, \pi_{\mathsf{bb}})$. It reads all “L1-vote”s from memory to form BB_1 and checks if $\mathsf{MerkleRoot}(\mathsf{BB}_1)$ equals μ_1 in the submitted transaction. It checks the submitted NIZK proof π_{bb}, and on success, adds (“fin-root”, $\mathsf{CT}, \mu_1, \mu_2$) to memory and moves state to $\mathsf{dispute}$.

$\mathsf{complain}(\mathsf{Q}, \psi)$: Contract checks if current state is $\mathsf{dispute}$ and time in $[T_5, T_6 - \Delta_1]$. It retrieves “fin-bb-bb” from memory into $\mathsf{pp}_{\mathsf{bb}}$, parses submitted ψ into (c', ρ), and verifies the receipt ρ. It then checks if c' is not in BB_1 (retrieved from memory), and if so, it adds (“complain”(Q, ψ)) to memory. If time is greater than T_7, it moves the state to $\mathsf{tallying}$.

$\mathsf{resolve}(\mathsf{S}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, \mathsf{Q}, \psi, \pi, \sigma)$: Contract checks if current state is $\mathsf{dispute}$ and time in $[T_5, T_7 - \Delta_1]$. If submitting party S is a $\mathbf{BBM}$ member and has a valid cert for vk, it checks S's signature on $(\mathsf{eID}, \text{“resolve”}, \mathsf{Q}, \psi, \pi)$. It retrieves μ_2 from memory, parses ψ into (c', ρ) and verifies the inclusion proof π using $\mathcal{F}_{\mathsf{BB}}.\mathsf{VerifyIncProof}$. If the proof verifies, (Q, ψ) is removed from memory. If time is greater than T_7, it moves the state to $\mathsf{tallying}$.

$\mathsf{tally}(\mathsf{P}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, \mathsf{tally}, \pi_{\mathsf{tally}}, \sigma)$: Contract checks if current state is $\mathsf{tallying}$ and time in $[T_8, T_9 - \Delta_1]$. If submitting party P is a $\mathbf{VC}$ member and has a valid cert for vk, it checks P's signature on $(\mathsf{eID}, \text{“tally”}, \mathsf{tally}, \pi_{\mathsf{tally}})$. It retrieves $\mathsf{pp}_{\mathsf{vc}}$ and CT from memory and verifies the proof π_{tally} using $\mathcal{F}_{\mathsf{DHE}}.\mathsf{VerifyDec}$. If the proof verifies, and there are no complaints in memory and the flag flag is not set, it adds (“fin-tally”, tally) to memory and moves the state to closed.

The relation $\mathcal{R}$ used to formulate $\mathcal{F}_{\mathsf{NIZK}}$ is a standard *algorithm* that is *internal* to $\mathcal{F}_{\mathsf{NIZK}}$. As such, an $\mathcal{F}_{\mathsf{NIZK}}$ wrt. a relation that makes *external* calls to other functionalities escapes this idealization.

We follow the work of Camenisch et al. [34], and introduce an intermediate *interactive* NIZK functionality $\mathcal{F}_{\mathsf{iNIZK}}$, where the functionality forwards the calls of the (now interactive) relation to the specified ideal functionality; see App. B-E for the formal definition.

We start by analyzing our protocols wrt. a globally verifiable $\mathcal{F}_{\mathsf{iNIZK}}$ (Step (1) in Fig. 3) as described above. To transition to using a standard $\mathcal{F}_{\mathsf{NIZK}}$ (and thus its realizations), we *inline* the external verification interface. Inlining the verification interface of an ideal functionality, however, would entail inlining an unspecified simulator, which makes little sense (verification calls are forwarded to an unspecified simulator). Thus, using the UCGS composition theorem, we first replace all external ideal functionalities with their concrete realizations (Step (2) in Fig. 3). Recall that this is made possible through globally verifiable variants of these external ideal functionalities.

These realizations are deterministic and stateless algorithms, which make the inlining meaningful (Step (3) in

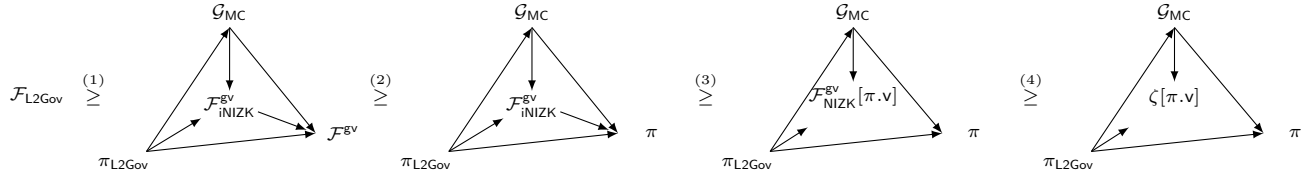

Figure 3. Structure of the UC proof of our layer-2 governance. $\mathcal{F}_{\mathsf{iNIZK}}$ is a NIZK with an interactive relation. $\mathcal{F}$ denotes other hybrids used ($\mathcal{F}_{\mathsf{DHE}}, \mathcal{F}_{\mathsf{BB}}, \mathcal{F}_{\mathsf{Sig}}, \mathcal{F}_{\mathsf{PKI}}$). Superscript '$\mathsf{gv}$' denotes 'globally verifiable' variants and 'v' denotes the verification interface. '$[]$' indicates in-lining. π is a realization of $\mathcal{F}$ and $\zeta[\pi.\mathsf{v}]$ is a NIZK realizing $\mathcal{F}_{\mathsf{NIZK}}$ wrt. a relation that uses the verification algorithms in π. '$\geq$' stands for 'is UC-realized by'.

Fig. 3). Indeed, it is not too hard to verify that the standard realizations of the verification interfaces of the hybrids that we use are deterministic and stateless. The check $\mathcal{F}_{\mathsf{BB}}.\mathsf{P}.\mathsf{Com}(\mathsf{BB}_2) = \mu_2$ is realized via Merkle root computation; $\mathcal{F}_{\mathsf{DHE}}.\mathsf{P}.\mathsf{Eval}(\mathsf{pp}_{\mathsf{vc}}, \mathsf{BB}_1 \cup \mathsf{BB}_2) = \mathsf{CT}$ is implemented through homomorphic evaluation of ElGamal ciphertexts; $\mathcal{F}_{\mathsf{Sig}}.\mathsf{P}.\mathsf{Verify}(\mathsf{vk}, (\mathsf{eID}, \text{"L1-vote"}, c), \sigma)$ runs the verification algorithm of a digital signature scheme and $\mathcal{F}_{\mathsf{DHE}}.\mathsf{P}.\mathsf{VerifyEnc}(\mathsf{pp}_{\mathsf{vc}}, c)$ runs a NIZK verification algorithm; and $\mathcal{F}_{\mathsf{PKI}}.\mathsf{P}.\mathsf{certVerify}(\mathsf{Q}, \mathsf{vk}, \mathsf{cert})$ runs the verification of the signatures issued by a PKI authority. (The interpretation of our proof for π_{DHE} with respect to $\mathcal{F}_{\mathsf{iNIZK}}$ whose relation calls the verification interface of $\mathcal{F}_{\mathsf{NIZK}}[\mathcal{R}_{\mathsf{AGG}}]$ is similar.)

Recall that ideal functionalities, and in particular $\mathcal{F}_{\mathsf{NIZK}}$, are not corruptible in UC. Hence, neither is the relation $\mathcal{R}$ run inside $\mathcal{F}_{\mathsf{NIZK}}$. This seems to create a discrepancy with using $\mathcal{F}_{\mathsf{iNIZK}}$: external relations are corruptible, while internalized ones are not. [34] relies on a further tweak to ideal functionalities, where the SID of $\mathcal{F}_{\mathsf{iNIZK}}$ specifies a special party that is incorruptible. This party is used by the interactive relation $\mathcal{R}$.

We follow a somewhat different approach that does away with this extra modification. The interactive relation $i\mathcal{R}$, compatibly with PID-wise connectivity, uses the party identifier P of the machine that calls $\mathcal{F}_{\mathsf{iNIZK}}.\mathsf{P}.\mathsf{Prove}$ or $\mathcal{F}_{\mathsf{iNIZK}}.\mathsf{P}.\mathsf{Verify}$. This, in particular, means that if P is honest, the external verification is also run honestly (due to our PID-wise corruption convention). The only gap that can arise is when a *corrupt* P places a call to the functionality, where in the inlined world the verification algorithm is run honestly, while in a world with externalized verification, a corrupted verification algorithm may be used. However, it is sufficient to note that the inlined functionality is at least as secure as (i.e., *UC-realizes*) the non-inlined (interactive) one. This is natural as the former is run wrt. an incorruptible, and thus more secure, relation. In more detail, for corrupted parties, the outputs are under the simulator's control, and all the simulator needs to do is to use the good verification algorithm when providing verification bits for the external (corrupted) verification. All in all, we do not need to rely on a special incorruptible party.

The final step (Step (4) in Fig. 3) is replacing $\mathcal{F}_{\mathsf{NIZK}}$ with a realization of it, which is made possible by the globally verifiable variant of $\mathcal{F}_{\mathsf{NIZK}}$ that we use.

When using globally verifiable functionalities in Step (1), the simulation of the *verification* interfaces requires a more nuanced approach than with local functionalities; otherwise the environment can distinguish between the real world and the ideal world. This is so because while in the real world all interfaces of the functionality follow the ideal functionality code, only the verification interface code of the ideal functionality is invoked. The simulator has to ensure the responses are still as if the other interfaces were also invoked. For instance, in $\mathcal{F}_{\mathsf{Sig}}$, if the environment requests to verify signatures produced via the Sign interface of the functionality, in the real world, they would always verify, independent of what the adversary returns for the verification bit. In the ideal world, though, the list of signatures Ver does not get populated. This means that if the simulator simply echoes the adversary's output bit, it will not be overwritten with 1. Thus, the simulator needs to make up for this gap by maintaining its own list of simulated signatures and then overwriting the adversary's bit to 1 if a (vk, m, σ) tuple appears on that list. Note, crucially, that since the list VK is also empty (as no keys can be generated by the environment for the session that $\mathsf{P}.\mathsf{Gen}$ is defined for), this bit will not be overwritten with 0.

We follow a similar strategy when using globally verifiable $\mathcal{F}_{\mathsf{NIZK}}$. Here, a further subtlety arises since to set the verification bit to 1 the simulator needs to output a valid *witness* for previously proven statements. The environment (via the adversary) may output an invalid witness for valid statements proven through the Prove interface. Thus, the simulator needs to keep track of and overwrite the adversary-provided witness. Although simulators in general need not know valid witnesses for proven statements, our simulators in both π_{DHE} and π_{L2Gov} do have this property.

We finally note that our UC analysis of π_{DHE} relies on the (multi-challenge) IND-CPA security of ElGamal. To this end, we need to embed the challenge public key and ciphertexts from the IND-CPA game into the simulator. This means that at some point in the analysis at least one secret key and the message/randomness pairs under the $\mathsf{P}.\mathsf{Encrypt}$ interface will not be known. This, however, does not pose a problem since at this point in the analysis, the reduction runs the entire UC experiment for the environment (which is the IND-CPA adversary) and thus can modify the specification of the globally verifiable functionalities as long as they remain indistinguishable from how they would behave when run with the simulator. In particular, it can directly respond with bit 1, even without a witness, as long as the statement is valid.

The challenges highlighted in this section and the specific

strategies used to overcome them are of broader relevance, particularly within the blockchain domain. In particular, similar approaches may be applied to the analysis of modular blockchain protocols where on-chain and off-chain components jointly interact with public and stateless verification interfaces. Such interfaces might involve the verification of threshold or multi-signatures, zero-knowledge proofs, commitment values, or Merkle proofs. Layer-2 protocols, such as rollups, are a prominent example in this category.

REFERENCES

[1] A. Kiayias and P. Lazos, "Sok: Blockchain governance," in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, ser. AFT '22. New York, NY, USA: Association for Computing Machinery, 2023, p. 61–73. [Online]. Available: https://doi.org/10.1145/3558535.3559794

[2] B. Zhang, R. Oliynykov, and H. Balogun, "A treasury system for cryptocurrencies: Enabling better collaborative intelligence," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019*. The Internet Society, 2019.

[3] B. Adida, "Helios: web-based open-audit voting," in *Proceedings of the 17th Conference on Security Symposium*, ser. SS'08. USA: USENIX Association, 2008, p. 335–348.

[4] A. Kiayias, T. Zacharias, and B. Zhang, "End-to-end verifiable elections in the standard model," in *Advances in Cryptology-EUROCRYPT 2015*. Springer, 2015, pp. 468–498.

[5] V. Cortier, P. Gaudry, and S. Glondu, "Belenios: a simple private and verifiable electronic voting system," in *Foundations of Security, Protocols, and Equational Reasoning: Essays Dedicated to Catherine A. Meadows*. Springer, 2019.

[6] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable, private smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1353–1370.

[7] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," http://plasma.io/plasma.pdf.

[8] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 136–145, full version: https://eprint.iacr.org/2000/067.

[9] J. Katz, "Round-optimal, fully secure distributed key generation," in *Annual International Cryptology Conference*. Springer, 2024, pp. 285–316.

[10] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2006, pp. 409–426.

[11] R. Canetti, A. Stoughton, and M. Varia, "Easyuc: Using easycrypt to mechanize proofs of universally composable security," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 2019, pp. 167–16 716.

[12] C. Abate, P. G. Haselwarter, E. Rivas, A. Van Muylder, T. Winterhalter, C. Hriţcu, K. Maillard, and B. Spitters, "Ssprove: A foundational framework for modular cryptographic proofs in coq," in *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, 2021, pp. 1–15.

[13] D. Bernhard, V. Cortier, D. Galindo, O. Pereira, and B. Warinschi, "Sok: A comprehensive analysis of game-based ballot privacy definitions," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 499–516.

[14] V. Cortier, D. Galindo, R. Küsters, J. Müller, and T. Truderung, "Sok: Verifiability notions for e-voting protocols," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 779–798.

[15] R. Küsters, T. Truderung, and A. Vogt, "Verifiability, privacy, and coercion-resistance: New insights from a case study," in *2011 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2011, pp. 538–553.

[16] C. Culnane and S. Schneider, "A peered bulletin board for robust use in verifiable voting systems," in *2014 IEEE 27th Computer Security Foundations Symposium*, 2014.

[17] N. Chondros, B. Zhang, T. Zacharias, P. Diamantopoulos, S. Maneas, C. Patsonakis, A. Delis, A. Kiayias, and M. Roussopoulos, "D-demos: A distributed, end-to-end verifiable, internet voting system," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016, pp. 711–720.

[18] M. Graf, R. Kusters, D. Rausch, S. Egger, M. Bechtold, and M. Flinspach, " Accountable Bulletin Boards: Definition and Provably Secure Implementation ," in *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*, 2024.

[19] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas, "Bitcoin as a transaction ledger: A composable treatment," *Journal of Cryptology*, vol. 37, no. 2, p. 18, 2024.

[20] S. Dziembowski, S. Faust, and K. Hostáková, "General state channel networks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018. [Online]. Available: https://doi.org/10.1145/3243734.3243856

[21] J. D. Cohen and M. J. Fischer, "A robust and verifiable cryptographically secure election scheme," in *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, 1985.

[22] D. L. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Commun. ACM*, vol. 24, no. 2, p. 84–90, 1981. [Online]. Available: https://doi.org/10.1145/358549.358563

[23] K. Sako and J. Kilian, "Receipt-free mix-type voting scheme," in *Advances in Cryptology — EUROCRYPT '95*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 393–403.

[24] D. Chaum, " Secret-Ballot Receipts: True Voter-Verifiable Elections ," *IEEE Security & Privacy*, vol. 2, no. 01, pp. 38–47, Jan. 2004. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MSECP.2004.1264852

[25] L. Hirschi, L. Schmid, and D. A. Basin, "Fixing the achilles heel of e-voting: The bulletin board," in *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 2021, pp. 1–17.

[26] Y. Ishai, R. Ostrovsky, and V. Zikas, "Secure multi-party computation with identifiable abort," in *Annual Cryptology Conference*. Springer, 2014, pp. 369–386.

[27] M. Chase, M. Kohlweiss, A. Lysyanskaya, and S. Meiklejohn, "Verifiable elections that scale for free," in *International Workshop on Public Key Cryptography*. Springer, 2013, pp. 479–496.

[28] B. Yu, J. K. Liu, A. Sakzad, S. Nepal, R. Steinfeld, P. Rimba, and M. H. Au, "Platform-independent secure blockchain-based voting system," in *International conference on information security*. Springer, 2018, pp. 369–386.

[29] T. Dimitriou, "Efficient, coercion-free and universally verifiable blockchain-based voting," *Computer Networks*, vol. 174, 2020.

[30] C. Spadafora, R. Longo, and M. Sala, "A coercion-resistant blockchain-based e-voting protocol with receipts," *Advances in Mathematics of Communications*, 2023.

[31] Z. Yin, B. Zhang, A. Nastenko, R. Oliynykov, and K. Ren, "A scalable coercion-resistant voting scheme for blockchain decision-making," *IEEE Transactions on Dependable and Secure Computing*, 2026.

[32] W. Choi, X. Liu, and V. Zikas, "Blockchain governance via sharp anonymous multisignatures," *Cryptology ePrint Archive*, 2023.

[33] J. Katz, U. Maurer, B. Tackmann, and V. Zikas, "Universally composable synchronous computation," in *Theory of Cryptography*, A. Sahai, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 477–498.

[34] J. Camenisch, M. Drijvers, and B. Tackmann, "Multi-protocol UC and its use for building modular and efficient protocols," Cryptology ePrint Archive, Report 2019/065, 2019. [Online]. Available: https://eprint.iacr.org/2019/065

[35] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," Cryptology ePrint Archive, Paper 2000/067, 2020, https://eprint.iacr.org/2000/067. [Online]. Available: https://eprint.iacr.org/2000/067

[36] C. Badertscher, R. Canetti, J. Hesse, B. Tackmann, and V. Zikas, "Universal composition with global subroutines: Capturing global setup within plain uc," in *Theory of Cryptography*, R. Pass and K. Pietrzak, Eds. Cham: Springer International Publishing, 2020, pp. 1–30.

[37] R. Canetti and T. Rabin, "Universal composition with joint state," in *Advances in Cryptology - CRYPTO 2003*, D. Boneh, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 265–281.

[38] R. Canetti, Y. Dodis, R. Pass, and S. Walfish, "Universally composable security with global setup," in *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007*, vol. 2006, 01 2006, p. 432.
[39] R. Cohen, J. Doerner, E. Lee, A. Lysyanskaya, and L. Roy, "An unstoppable ideal functionality for signatures and a modular analysis of the dolev-strong broadcast," Cryptology ePrint Archive, Report 2024/1807, 2024. [Online]. Available: https://eprint.iacr.org/2024/1807
[40] R. Canetti, D. Shahaf, and M. Vald, "Universally composable authentication and key-exchange with global pki," in *Public-Key Cryptography–PKC 2016: 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part II 19*. Springer, 2016, pp. 265–296.

APPENDIX A
THE UC FRAMEWORK

The UC framework provides strong security guarantees in the sense that UC-secure protocols can be used securely and modularly in a wide variety of contexts. In a nutshell, UC compares the execution of a real protocol in the presence of an adversary to that of an ideal process in the presence of another adversary. If for any adversary against the real protocol there exists an ideal adversary (aka. a simulator) against the ideal process so that no interactive distinguisher (aka. the environment) can distinguish the two, the real protocol is said to UC-realize the ideal process.

Using this definition of security, one can first design a (higher-level) protocol with respect to a simpler ideal process, and then replace the ideal process with a real process that UC-realizes it. This follows from the UC composition theorem, the proof of which relies on the fact that the effects of any higher-level protocol can be absorbed into the environment. This replacement process can be extended to multiple ideal processes and enables a modular treatment of security for complex protocols. It is thus particularly suitable for our goals.

Although the core idea is arguably natural, defining a formal execution model that is sufficiently expressive to capture real-world protocols of interest turns out to be technical. For instance, one needs to delineate the boundaries of protocol instances, appropriately model shared resources, consider corrupted parties, and constrain the environment so that security is both feasible and meaningful.

We give a brief overview of the UC framework and refer to [35, Section 2] and [36, Section 2] for further details.

ITMS, ITIS, AND PROTOCOLS. The UC framework models protocols as a system of *interactive Turing machines* (ITMs) that interact through input, output, and backdoor tapes. These tapes respectively hold inputs from the calling ITM, return values for the calling ITM, and allow interactions with an adversary, which is also modeled as an ITM. An ITM *instance* (ITI) is specified by a tuple $(\mu, \mathsf{pid}, \mathsf{sid})$, where μ specifies how parties should interact on various inputs, pid is a party identifier (PID), and sid is a session identifier (SID). The interpretation is that ITIs with the same PID run on a single machine (or more generally on a cluster of local machines requiring no network connectivity). The SID models independent instances of the same protocol π. All ITIs have unique identities of the form $(\mathsf{pid}, \mathsf{sid})$. A protocol π is a *set* of *well-connected* ITIs: if an ITI can pass inputs to another ITI, then the latter can pass outputs back to it. Moreover, we require that this connectivity respects PIDs, except for ideal functionalities, the environment, and the adversary. We will discuss these below. The *main* machines of a protocol are the ITIs that receive inputs from ITIs outside the protocol.

THE ENVIRONMENT AND THE ADVERSARY. The environment $\mathcal{E}$ and the adversary $\mathcal{A}$ (which is called the *simulator* when run in an 'ideal world') are ITIs that have special pid's and no sid. The environment models honest parties in higher-level protocols (including protocols in other sessions); the adversary captures adversarial influence on the protocol (including that of corrupted parties that come under its control). The environment can freely interact with the adversary. The adversary can communicate with all other machines via their backdoor tapes.

CORRUPTIONS. The environment may instruct the adversary to *corrupt* a party with PID pid that must differ from those of $\mathcal{A}$, $\mathcal{E}$, and any ideal functionality. As a result, the contents of the internal work tapes of *all* ITIs with PID pid are sent to the adversary. Moreover, from now on, these ITIs act as 'forwarders': they forward all their communication to $\mathcal{A}$ and receive instructions from it to respond to other machines. Thus, our choice of corruption corresponds to PID-wise full Byzantine corruptions. In this work, we consider only (fully) static corruptions, meaning that $\mathcal{E}$ instructs $\mathcal{A}$ to corrupt parties at the onset of execution, before any other interaction. Throughout, we let $\mathbf{C}$ denote the set of corrupt parties, and $\mathbf{H}$ the set of honest parties, i.e., those that are not corrupted.

IDEAL FUNCTIONALITIES AND IDEAL PROTOCOLS. An *ideal functionality* is an ITI that models a 'trusted party' carrying out a specific task. The ideal protocol $\mathsf{IDEAL}(\mathcal{F})$ for an ideal functionality $\mathcal{F}$ is $\mathcal{F}$ attached to a number of *dummy parties* that simply forward the communication to and from $\mathcal{F}$. This gives the higher-level protocols or the environment the impression of interacting with a distributed protocol (rather than a single machine). By slight abuse of notation, we will use $\mathcal{F}$ for $\mathsf{IDEAL}(\mathcal{F})$. Note that through ideal functionalities, different parties gain access to shared state. As mentioned above, ideal functionalities are *incorruptible*. The adversary can still interact with ideal functionalities both through the backdoor tape and on behalf of the corrupted dummy parties.

EXECUTIONS. The execution of a protocol π with an adversary $\mathcal{A}$ and an environment $\mathcal{E}$ on advice string $z \in \{0,1\}^*$ starts by running $\mathcal{E}(z)$. The environment can then send messages to other machines *on behalf of other ITIs*. (This is required so that $\mathcal{E}$ can model higher-level protocols and thus allow a composition theorem to be established.) The environment can only pass inputs to the *main machines* of a protocol π and can only assume *honest external* identities that do not appear in π. The adversary, on the other hand, can assume the identities of any corrupted party in any session. The environment's calls must also respect PID-wise connectivity. The execution terminates when $\mathcal{E}$ returns a bit, which we denote by $\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(z)$. We emphasize that connectivity

depends only on main machines and external identities, and not on the internal structure of a protocol. Furthermore, this connectivity is fixed at the onset.

UC EMULATION. We say that a protocol π UC-emulates another (typically more idealized) protocol ϕ if for any adversary $\mathcal{A}$ there exists a *simulator* $\mathcal{S}$ such that for all environments $\mathcal{E}$ and all $z \in \{0,1\}^*$:

$$\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(z) \approx \mathsf{EXEC}_{\phi,\mathcal{S},\mathcal{E}}(z) \ .$$

Here '$\approx$' denotes computational indistinguishability of the output bits.

THE UC THEOREM. Let ρ, ϕ, π be protocols (sets of ITIs) such that $\phi \subseteq \rho$, i.e., ρ (potentially) calls protocol ϕ. Suppose that π and ϕ have compatible external identities (and thus main machines), and that π and $\rho \setminus \phi$ have disjoint identities. Let $\rho^{\phi \to \pi}$ be a protocol that is identical to ρ except that all instances of the (typically ideal) functionality ϕ are replaced by instances of the (more real) protocol π. Canetti [35] presents a composition theorem stating that if π UC-emulates ϕ, then $\rho^{\phi \to \pi}$ UC-emulates ρ. When defining UC-emulation, the machines of ϕ replaced by $\rho^{\phi \to \pi}$ are sometimes referred to as the *target protocol*. One often designs protocols so that the main machines of the target protocol instance share a common SID called the *target session*.

A. UC with shared subroutines

Shared subroutines, e.g., blockchains and synchronized clocks, are protocols that may be used by several other protocols. Unlike a (local) functionality, a single instance of a global functionality may take inputs from, and provide outputs to, multiple instances of (unknown) protocols. Thus, such a functionality can be in any state when it is accessed by some protocol. The first attempt to augment the UC framework to capture such shared setups was the Universal Composition with Joint State (JUC) [37] model. Canetti et al. [38] proposed UC with Global Setups, often referred to as the GUC framework. Badertscher et al. [36] build on the latest iteration of the UC framework [35] to model global functionalities as shared subroutines within the (plain) UC framework. Using the UCGS framework, one thus avoids complex compositional frameworks such as [37], [38].

The (simple) UCGS framework, where the machines and their connectivities are fixed at the onset, treats a global protocol γ as a sub-protocol of $\pi \cup \gamma$ such that the main machines of γ are a subset of the main machines of $\pi \cup \gamma$. This allows γ to be accessed by higher-level protocols.

THE (SIMPLE) UCGS THEOREM. Let ρ, ϕ, π, γ be such that ρ, $\phi \cup \gamma$, and $\pi \cup \gamma$ are protocols and $\phi \cup \gamma \subseteq \rho$. Suppose that $\phi \cup \gamma$ and $\pi \cup \gamma$ have compatible sets of identities and that π and $\rho \setminus \phi$ have disjoint identities. Suppose $\pi \cup \gamma$ UC-emulates $\phi \cup \gamma$. Then $\rho^{\phi \to \pi}$ UC-emulates ρ.

Note that the prerequisite is UC-emulation in the presence of γ, and that $\rho^{\phi \to \pi}$ does not replace γ. In this formulation, the UCGS theorem allows secure composition even if the machines of π and γ communicate freely. This is because the (plain) UC theorem allows the main machines to also communicate freely.

Looking ahead, certain cryptographic artifacts such as proofs, signatures, and commitments must be verified by distinct components of our governance solution. To this end, we use ideal functionalities, some of whose interfaces are shared across different sessions. In Sect. VII, we present the technical UC challenges that arise in this treatment. The discussion in this section is general and applicable to protocols that follow similar designs.

B. Further syntax

In our specifications of ideal functionalities, we sometimes need to *sanitize* the simulator's outputs to ensure that attacks do not have an overly adverse effect on the functionality when used as a hybrid. For instance, in honest-majority settings, we often sanitize a $\perp$-output by the simulator to a non-$\perp$ value so that liveness is preserved. We denote such a sanitization process by $\mathsf{San}(v) : \mathrm{Cond}(v)$, meaning that a sanitization procedure, on input v, outputs a possibly new value for v guaranteed to satisfy the predicate $\mathrm{Cond}(v)$ if the condition was not already met. The specific ways in which $\mathsf{San}()$ is implemented, beyond adherence to basic requirements such as polynomial runtime, are not critical for us; all that matters is that $\mathrm{Cond}(v)$ is met so that the hybrid is useful. Such sanitization procedures have been used in [39] to define an available signature ideal functionality.

APPENDIX B
STANDARD HYBRIDS

A. $\mathcal{F}_{\mathsf{AC}}$: Authenticated Channels

$\mathcal{F}_{\mathsf{AC}}$ (Func. 5) idealizes an authenticated channel (without confidentiality) where a party P can send a message x to a party Q, who can later fetch it, together with the sender's id P. Messages are guaranteed authentic (i.e., in fact sent by the party whose id is retrieved) and delivered at most once, though they may be delayed by up to Δ rounds and may arrive out of order. Send allows party P to send a message x to a party Q. The sending round is recorded and later used to compute delays. The simulator sees the message content and the destination id. For simplicity, we assume Send supports arbitrary-length messages x. Fetch allows a party to fetch messages sent to it (from any party). Messages are returned together with their sender ids. The simulator decides which pending messages are delivered on a given fetch; once a message has been pending for Δ rounds its delivery is forced, so messages can be delayed by at most Δ rounds. Note that the simulator is *not* able to inject any messages on behalf of any honest party. The sanitizer San ensures that Sim only picks messages whose destinations match the receiver's identity, filtering out any other indices it supplies.

Functionality 5: $\mathcal{F}_{\mathsf{AC}}[\mathbf{P}, \Delta]$

Parameters
1: Hybrids: $\mathcal{G}_{\mathsf{Clock}}[\mathbf{P}']$
2: Requires: $\mathbf{P} \subseteq \mathbf{P}'$

Initialize()
3: $\mathrm{ctr} \leftarrow 0$; $\mathbf{L} \leftarrow []$

P.Send(Q, x) for $\mathsf{P} \in \mathbf{P}$
4: $t \leftarrow \mathsf{T}()$
5: $\mathrm{ctr} \leftarrow \mathrm{ctr} + 1$
6: $\mathbf{L}[\mathrm{ctr}] \leftarrow (\mathsf{P}, \mathsf{Q}, t, x)$
7: Sim(P.Send, $(\mathrm{ctr}, \mathsf{Q}, t, x)$)
8: **return** ok

P.Fetch() for $\mathsf{P} \in \mathbf{P}$
9: $t \leftarrow \mathsf{T}()$
10: $\mathbf{I}_{\mathsf{late}} \leftarrow [i \in [\mathrm{ctr}] : \mathbf{L}[i] = (*, \mathsf{P}, t_i, *) \wedge t - t_i \geq \Delta]$
11: $\mathbf{I}_{\mathsf{early}} \leftarrow \mathsf{Sim}(\mathsf{P.Fetch})$
12: $\mathsf{San}(\mathbf{I}_{\mathsf{early}})$: $(\forall i \in \mathbf{I}_{\mathsf{early}})\mathbf{L}[i] = (*, \mathsf{P}, *, *)$
13: $\mathbf{Msgs} \leftarrow [\mathbf{L}[i] : i \in \mathbf{I}_{\mathsf{late}} \cup \mathbf{I}_{\mathsf{early}}]$
14: **for** $i \in \mathbf{I}_{\mathsf{late}} \cup \mathbf{I}_{\mathsf{early}}$ **do** $\mathbf{L}[i] \leftarrow \perp$
15: **return** $\mathbf{Msgs}$

B. $\mathcal{F}_{\mathsf{SC}}$: Secure Channels

$\mathcal{F}_{\mathsf{SC}}$ (Func. 6) is similar to the authenticated channels functionality $\mathcal{F}_{\mathsf{AC}}$, except that it leaks only the length of messages to the simulator.

C. $\mathcal{F}_{\mathsf{BC}}$: Broadcast Channels

$\mathcal{F}_{\mathsf{BC}}$ (Functionality 7) idealizes a one-time broadcast channel. Interface Cast allows a designated party P^* to broadcast a message x once to all other parties. The sending round is recorded and later used to compute delays. Interface Fetch allows any party to fetch the single broadcast message. The guarantee is that the same message x is always delivered to the fetching party Δ rounds after it was cast.

D. $\mathcal{F}_{\mathsf{CRS}}$: Common Reference String

This functionality (Func. 8) allows all parties to access a common string, generated by a string generation algorithm D.

E. $\mathcal{F}_{\mathsf{NIZK}}$: Non-Interactive Zero-Knowledge

$\mathcal{F}_{\mathsf{NIZK}}$ (Func. 9) idealizes a non-interactive zero-knowledge proof system for a relation R. Interface Prove allows parties to produce proofs, chosen by the simulator, for an instance x with a witness w satisfying R. The Ver table maintains consistency. The sanitizer ensures that the output proofs always verify.

Interface Verify allows verifying proofs, where a simulator outputs a witness w, which is then used to compute the result via R. Note that this corresponds to a *proof of knowledge* property where, in a realization, the simulator is implemented via a knowledge extractor. Func. 10 idealizes an interactive zero-knowledge proof system. It is similar to the non-interactive functionality, but instead of checking the relation R locally, it relies on another functionality $\mathcal{F}_{\mathsf{R}}$. $\mathcal{F}_{\mathsf{R}}$ comes with a single check interface that takes (x, w) and returns 0 or 1.

F. $\mathcal{F}_{\mathsf{Sig}}$: Digital Signatures

This functionality (Func. 11) idealizes a public-key signature scheme. The simulators are "responsive" since this functionality models non-interactive local signature schemes. Interface Gen allows parties to generate a key. The verification key is picked by the simulator and sanitized to differ from existing vk's, including those for which a *valid* entry in Ver exists (i.e., "pre-forgeries" are avoided). Interface Sign allows a party to generate a signature on a given message. The signature is sanitized so that it is always available (a consequence of locality) and not previously marked invalid. Interface Verify allows verifying signatures under any verification key (we do not assume that verification keys are authenticated and available; hence a vk must be provided). This interface guarantees that no new signature under an honestly generated key verifies (unless the party has been corrupted). Therefore, signatures are strongly unforgeable.

G. $\mathcal{F}_{\mathsf{PKI}}$: Public-Key Infrastructure

$\mathcal{F}_{\mathsf{PKI}}$ (Func. 12) idealizes a PKI and allows parties to register their public keys and, as a result, obtain a certificate, which any party can verify. In more detail, Setup allows a party to generate the root-of-trust public parameters (essentially, the PKI's verification key). Register lets parties register a key vk of their choice once, thereby obtaining a certificate cert. Interface CertVerify allows any party to verify whether a certificate cert for a key vk and party Q is valid. Retrieve allows parties to obtain the key registered by some other party Q. Func. 12 is based on that in [40].

H. $\mathcal{G}_{\mathsf{Clock}}$: Shared Clocks

We discussed $\mathcal{G}_{\mathsf{Clock}}$ and the timing mechanism in the main body. Here we present the details of the functionality (Func. 13). Register allows parties in a particular session to register with the clock functionality. DeRegister allows parties to de-register from the clock functionality. (We do not use DeRegister in this work and assume all honest parties are online.) OK allows parties in a session to indicate that they are done with the current round. T allows parties in any session, and the simulator, to obtain the current round number.

Parties in our protocols keep track of tasks already completed for a given round via an internal variable. Once all tasks for a round are completed, the party calls OK. To declutter the protocol descriptions, we omit explicit OK-ing of the clock.

The clock functionality allows a higher-level protocol to operate at the same round as a lower-level one, as long as this higher-level protocol also registers with the clock: the round

Functionality 6: $\mathcal{F}_{\mathsf{SC}}[\mathbf{P}, \Delta]$

Parameters
1: Hybrids: $\mathcal{G}_{\mathsf{Clock}}[\mathbf{P}']$
2: Requires: $\mathbf{P} \subseteq \mathbf{P}'$

Initialize()
3: $\mathrm{ctr} \leftarrow 0$; $\mathbf{L} \leftarrow [\,]$

P.Send(Q, x) for $\mathsf{P} \in \mathbf{P}$
4: $t \leftarrow \mathsf{T}()$
5: $\mathrm{ctr} \leftarrow \mathrm{ctr} + 1$
6: $\mathbf{L}[\mathrm{ctr}] \leftarrow (\mathsf{P}, \mathsf{Q}, t, x)$
7: Sim(P.Send, $(\mathrm{ctr}, \mathsf{Q}, t, 1^{|x|})$)
8: **return** ok

P.Fetch() for $\mathsf{P} \in \mathbf{P}$
9: $t \leftarrow \mathsf{T}()$
10: $\mathbf{I}_{\mathsf{late}} \leftarrow [i \in [\mathrm{ctr}] : \mathbf{L}[i] = (*, \mathsf{P}, t_i, *) \wedge t - t_i \geq \Delta]$
11: $\mathbf{I}_{\mathsf{early}} \leftarrow$ Sim(P.Fetch)
12: San($\mathbf{I}_{\mathsf{early}}$): $(\forall i \in \mathbf{I}_{\mathsf{early}})\mathbf{L}[i] = (*, \mathsf{P}, *, *)$
13: $\mathbf{Msgs} \leftarrow [\mathbf{L}[i] : i \in \mathbf{I}_{\mathsf{late}} \cup \mathbf{I}_{\mathsf{early}}]$
14: **for** $i \in \mathbf{I}_{\mathsf{late}} \cup \mathbf{I}_{\mathsf{early}}$ **do** $\mathbf{L}[i] \leftarrow \perp$
15: **return** $\mathbf{Msgs}$

Functionality 7: $\mathcal{F}_{\mathsf{BC}}[\mathsf{P}^*, \mathbf{P}, \Delta]$

Parameters
1: Hybrids: $\mathcal{G}_{\mathsf{Clock}}[\mathbf{P}']$
2: Requires: $\mathsf{P}^* \in \mathbf{P} \wedge \mathbf{P} \subseteq \mathbf{P}'$

Initialize()
3: $m^* \leftarrow \perp$; $X \leftarrow \perp$

P.Cast(m) for $\mathsf{P} = \mathsf{P}^*$, once
4: **if** $X \neq \perp$ **then return** $\perp$
5: $t \leftarrow \mathsf{T}()$
6: **if** $m \neq \perp$ **then** $X \leftarrow (t, m)$
7: Sim(P.Cast, (t, m))
8: **return** ok

P.Fetch() for $\mathsf{P} \in \mathbf{P}$
9: $(t, m) \leftarrow \mathsf{parse}(X)$
10: $m' \leftarrow$ Sim(P.Fetch)
11: **if** $\mathsf{P}^* \notin \mathbf{H}$ **then** $m \leftarrow m'$
12: **if** $\mathsf{T}() - t \geq \Delta \wedge m^* = \perp$ **then** $m^* \leftarrow m$
13: **return** m^*

Functionality 8: $\mathcal{F}_{\mathsf{CRS}}[\mathbf{P}, D]$

Parameters
1: Hybrids: None
2: Requires: None

Initialize()
3: crs $\leftarrow D()$

P.Gen() for $\mathsf{P} \in \mathbf{P}$
4: Sim(P.Gen, crs)
5: **return** crs

Functionality 9: $\mathcal{F}_{\mathsf{NIZK}}[\mathbf{P}, \mathsf{R}]$

Parameters
1: Hybrids: None
2: Requires: None

Initialize()
3: $\mathsf{Ver}[*, *] \leftarrow \perp$

P.Prove(x, w) for $\mathsf{P} \in \mathbf{P}$
4: **check** $\mathsf{R}(x, w)$
5: $\pi \leftarrow$ Sim(P.Prove, x)
6: San(π): $\pi \neq \perp \wedge \mathsf{Ver}[x, \pi] \neq 0$
7: $\mathsf{Ver}[x, \pi] \leftarrow 1$
8: **return** π

P.Verify(x, π) for $\mathsf{P} \in \mathbf{P}$; if gv: any sid
9: $w \leftarrow$ Sim(P.Verify, x, π)
10: **if** $\mathsf{Ver}[x, \pi] \neq \perp$ **then return** $\mathsf{Ver}[x, \pi]$
11: $\mathsf{Ver}[x, \pi] \leftarrow \mathsf{R}(x, w)$
12: **return** $\mathsf{Ver}[x, \pi]$

only advances once that higher-level protocol also indicates it is ready to proceed. Parties and functionalities can de-register from the clock, so not all users need to engage in protocol actions (such as voters in voting, complaints, etc.) for a round to advance. In this work, however, we assume all parties are online and registered with the clock.

In our protocol descriptions, rounds are relative in that a protocol/functionality obtains the current round at an initialization stage and proceeds relative to this initial round number with various delay parameters (typically denoted Δ) in subsequent rounds.

The round complexities of some of our protocols (e.g., governance when called on complaints) are higher than those needed for their realization. This is to give parties a "window of opportunity" to take certain actions (e.g., file a complaint).

We do not distinguish between $\perp$ as an error symbol versus a symbol indicating that protocol output is not yet ready. The understanding is that protocol descriptions are public, and

Functionality 10: $\mathcal{F}_{\mathsf{iNIZK}}[\mathbf{P}]$

Parameters
1: Hybrids: $\mathcal{F}_{\mathsf{R}}[\mathbf{P}']$
2: Requires: $\mathbf{P} \subseteq \mathbf{P}'$

Initialize()
3: $\mathsf{Ver}[*,*] \leftarrow \bot$

P.Prove(x, w) for $\mathsf{P} \in \mathbf{P}$
4: **check** $\mathcal{F}_{\mathsf{R}}$.P.Check(x, w)
5: $\pi \leftarrow \mathsf{Sim}(\mathsf{P.Prove}, x)$
6: $\mathsf{San}(\pi)$: $\pi \neq \bot \wedge \mathsf{Ver}[x, \pi] \neq 0$
7: $\mathsf{Ver}[x, \pi] \leftarrow 1$
8: **return** π

P.Verify(x, π) for $\mathsf{P} \in \mathbf{P}$; if gv: any sid
9: $w \leftarrow \mathsf{Sim}(\mathsf{P.Verify}, x, \pi)$
10: **if** $\mathsf{Ver}[x, \pi] \neq \bot$ **then return** $\mathsf{Ver}[x, \pi]$
11: $\mathsf{Ver}[x, \pi] \leftarrow \mathcal{F}_{\mathsf{R}}$.P.Check$(x, w)$
12: **return** $\mathsf{Ver}[x, \pi]$

Functionality 11: $\mathcal{F}_{\mathsf{Sig}}[\mathbf{P}, \mathbf{S}]$

Parameters
1: Hybrids: None
2: Requires: None

Initialize()
3: $\mathsf{VK}[*] \leftarrow \bot$; $\mathsf{Ver}[*,*,*] \leftarrow \bot$

P.Gen() for $\mathsf{P} \in \mathbf{S}$
4: **if** $\mathsf{VK}[\mathsf{P}] \neq \bot$ **then return** $\mathsf{VK}[\mathsf{P}]$
5: $\mathsf{vk} \leftarrow \mathsf{Sim}(\mathsf{P.Gen})$
6: $\mathsf{San}(\mathsf{vk})$: $\mathsf{vk} \neq \bot \wedge \mathsf{vk} \notin \mathsf{VK} \wedge \mathsf{Ver}[\mathsf{vk}, *, *] \neq 1$
7: $\mathsf{VK}[\mathsf{P}] \leftarrow \mathsf{vk}$
8: **return** $\mathsf{VK}[\mathsf{P}]$

P.Sign(m) for $\mathsf{P} \in \mathbf{S}$
9: **check** $\mathsf{VK}[\mathsf{P}] \neq \bot$
10: $\mathsf{vk} \leftarrow \mathsf{VK}[\mathsf{P}]$
11: $\sigma \leftarrow \mathsf{Sim}(\mathsf{P.Sign}, m)$
12: $\mathsf{San}(\sigma)$: $\sigma \neq \bot \wedge \mathsf{Ver}[\mathsf{vk}, m, \sigma] \neq 0$
13: $\mathsf{Ver}[\mathsf{vk}, m, \sigma] \leftarrow 1$
14: **return** σ

P.Verify(vk, m, σ) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{S}$; if gv: any sid
15: **if** $\mathsf{Ver}[\mathsf{vk}, m, \sigma] \neq \bot$ **then return** $\mathsf{Ver}[\mathsf{vk}, m, \sigma]$
16: $b \leftarrow \mathsf{Sim}(\mathsf{P.Verify}, \mathsf{vk}, m, \sigma)$
17: $\mathsf{San}(b)$: $b \in \{0, 1\}$
18: **if** $\exists \mathsf{P}' \in \mathbf{H}.\, \mathsf{vk} = \mathsf{VK}[\mathsf{P}']$ **then** $b \leftarrow 0$ // unique vk per P
19: $\mathsf{Ver}[\mathsf{vk}, m, \sigma] \leftarrow b$
20: **return** b

Functionality 12: $\mathcal{F}_{\mathsf{PKI}}[\mathbf{P}]$

Parameters
1: Hybrids: None
2: Requires: None

Initialize()
3: $\mathsf{pp}^* \leftarrow \bot$; $\mathsf{Reg}[*,*] \leftarrow \bot$; $\mathsf{Ver}[*,*,*,*] \leftarrow \bot$

P.Setup() for $\mathsf{P} \in \mathbf{P}$
4: **if** $(\mathsf{pp}^* \neq \bot)$ **then return** pp^*
5: $\mathsf{pp}^* \leftarrow \mathsf{Sim}(\mathsf{P.Setup})$
6: $\mathsf{San}(\mathsf{pp}^*)$: $\mathsf{pp}^* \neq \bot$
7: **return** pp^*

P.Register(pp, vk) for $\mathsf{P} \in \mathbf{P}$
8: **check** $\mathsf{pp} = \mathsf{pp}^* \wedge \mathsf{pp}^* \neq \bot$
9: $\mathsf{cert} \leftarrow \mathsf{Sim}(\mathsf{P.Register}, \mathsf{pp}, \mathsf{vk})$
10: $\mathsf{San}(\mathsf{cert})$: $\mathsf{cert} \neq \bot$
11: **if** $\mathsf{Reg}[\mathsf{pp}, \mathsf{P}] \neq \bot$ **then return** $\bot$
12: $\mathsf{Reg}[\mathsf{pp}, \mathsf{P}] \leftarrow \mathsf{vk}$
13: $\mathsf{Ver}[\mathsf{pp}, \mathsf{P}, \mathsf{vk}, \mathsf{cert}] \leftarrow 1$
14: **return** cert

P.CertVerify(pp, Q, vk, cert) for $\mathsf{P} \in \mathbf{P}$; if gv: any sid
15: **if** $\mathsf{Ver}[\mathsf{pp}, \mathsf{Q}, \mathsf{vk}, \mathsf{cert}] \neq \bot$ **then return** $\mathsf{Ver}[\mathsf{pp}, \mathsf{Q}, \mathsf{vk}, \mathsf{cert}]$
16: $b \leftarrow \mathsf{Sim}(\mathsf{P.CertVerify}, \mathsf{pp}, \mathsf{Q}, \mathsf{vk}, \mathsf{cert})$
17: $\mathsf{San}(b)$: $b \in \{0, 1\}$
18: **if** $\mathsf{pp} = \mathsf{pp}^*$ **then return** $(\mathsf{Ver}[\mathsf{pp}, \mathsf{Q}, \mathsf{vk}, \mathsf{cert}] = 1)$
19: $\mathsf{Ver}[\mathsf{pp}, \mathsf{Q}, \mathsf{vk}, \mathsf{cert}] \leftarrow b$
20: **return** b

P.Retrieve(pp, Q) for $\mathsf{P} \in \mathbf{P}$
21: $\mathsf{Sim}(\mathsf{P.Retrieve}, \mathsf{pp}, \mathsf{Q})$
22: **return** $\mathsf{Reg}[\mathsf{pp}, \mathsf{Q}]$

hence so are the implicit round numbers and time frames. Thus, depending on the current round number, $\bot$ can either be interpreted as an "error" or "not yet ready."

In calling $\mathcal{G}_{\mathsf{Clock}}$ (and later $\mathcal{G}_{\mathsf{MC}}$), local protocols and functionalities use their session-id, which is typically omitted from the parameter list.

APPENDIX C
DETAILS OF $\mathcal{G}_{\mathsf{MC}}$

We introduce a new ideal blockchain functionality that, while relatively simple, supports smart contracts and is a close model of a real blockchain. Existing idealizations are fine-tuned to real-world realizations [19], making them cumbersome to use as hybrids, or else are too idealistic [20].

Our mainchain functionality $\mathcal{G}_{\mathsf{MC}}$ comes with two interfaces: a Submit interface that allows parties to submit transactions tx to the ledger. The transaction and the current time/round

Functionality 13: $\mathcal{G}_{\mathsf{Clock}}[\mathbf{P}]$

Parameters
1: Hybrids: None
2: Requires: None

Initialize()
3: $r \leftarrow 1$; $\mathsf{Reg} \leftarrow \varnothing$; $\mathsf{flag} \leftarrow \bot$; $\mathbf{D}[*] \leftarrow \bot$

$\mathsf{P}_{\mathsf{sid}}$.Register() for $\mathsf{P} \in \mathbf{P}$, any sid
4: **if** $\mathsf{P}_{\mathsf{sid}} \notin \mathsf{Reg}$ **then** $\mathbf{D}[\mathsf{P}_{\mathsf{sid}}] \leftarrow \bot$
5: $\mathsf{Reg} \leftarrow \mathsf{Reg} \cup \{\mathsf{P}_{\mathsf{sid}}\}$
6: **return** ok

$\mathsf{P}_{\mathsf{sid}}$.DeRegister() for $\mathsf{P} \in \mathbf{P}$, any sid
7: $\mathsf{Reg} \leftarrow \mathsf{Reg} \setminus \{\mathsf{P}_{\mathsf{sid}}\}$
8: **return** ok

$\mathsf{P}_{\mathsf{sid}}$.T() for $\mathsf{P} \in \mathbf{P} \cup \{\mathsf{Sim}\}$, any sid
9: **return** r

$\mathsf{P}_{\mathsf{sid}}$.OK() for $\mathsf{P} \in \mathbf{P}$, any sid
10: **if** $\mathsf{P}_{\mathsf{sid}} \notin \mathsf{Reg}$ **then return** $\bot$
11: $\mathbf{D}[\mathsf{P}_{\mathsf{sid}}] \leftarrow \top$
12: $\mathsf{flag} \leftarrow \top$
13: **for** $\mathsf{P}'_{\mathsf{sid}'} \in \mathsf{Reg} \setminus \mathbf{C}$ **do**
14: **if** $\mathbf{D}[\mathsf{P}'_{\mathsf{sid}'}] = \bot$ **then** $\mathsf{flag} \leftarrow \bot$
15: **if** $\mathsf{flag} = \top$ **then**
16: $r \leftarrow r + 1$
17: **for** $\mathsf{P}'_{\mathsf{sid}'} \in \mathsf{Reg} \setminus \mathbf{C}$ **do** $\mathbf{D}[\mathsf{P}'_{\mathsf{sid}'}] \leftarrow \bot$
18: $\mathsf{Sim}(\mathsf{P}_{\mathsf{sid}}.\mathsf{OK})$
19: **return** ok

are added to a buffer. A Read interface allows parties to read the current state of the ledger. When a party reads, the functionality populates the log with entries from the buffer. All transactions at least Δ_{lat} rounds old (where Δ_{lat} is the latency parameter) are included in the log. Read then returns a prefix of this log according to a pointer for each party P that calls it. The pointer is set by the simulator, but is sanitized so that parties see a sufficiently recent prefix of the log. In more detail, a second slack parameter Δ_{slack} decides how far back from the current round the pointer can be, and thus how synchronized parties are. The pointer is guaranteed never to decrease.

Our governance protocol relies on deploying smart contracts. At the core of our formalization is a concrete validity predicate $\mathsf{Val}_{\mathsf{MC}}$. At a high level, the $\mathsf{Val}_{\mathsf{MC}}$ predicate interprets transactions as code with arguments and evaluates the code on the current contract state. This is done iteratively starting from the genesis block. The code can specify calling other deployed contracts or transferring funds.

More precisely, the *contract state* CS is a key-value store that stores, for each contract-id cID, a tuple $(\mathsf{code}, \mathsf{mem}, \mathsf{funds})$ of the contract's code, memory, and funds. Predicate $\mathsf{Val}_{\mathsf{MC}}$ relies on an UpdateCS procedure that, given a contract state CS and a transaction tx, outputs a tuple containing an updated contract state and a return value (or $\bot$, indicating an execution error or an invalid transaction). The UpdateCS procedure is applied iteratively to each transaction in a given log to compute the final contract state for the whole log.

The iterative process starts with an initialization procedure InitCS that reads off the genesis block, a parameter of our functionality, to produce the initial contract state. For example, this can assign certain contracts their funds as specified in the genesis block. Thereafter, UpdateCS parses a given transaction tx as $(\mathsf{cID}, \mathsf{call}, \mathsf{args}; v, \mathsf{sID})$ comprising a contract ID, a method `call`, arguments args, a received-value amount v, and a fund-sender contract-id sID. Before passing a top-level transaction to UpdateCS, ContractState enforces isolation by checking that $v = 0$ and $\mathsf{sID} = \varepsilon$: this prevents external users from spoofing internal cross-contract calls or injecting unbacked funds, restricting value transfers to internal sub-calls between already-funded contracts. If the specified method is `install`, UpdateCS parses args further as code and initializes $\mathsf{CS}[\mathsf{cID}]$ with the code, an unset memory state, and a balance of 0. (If a contract is already deployed at cID, indicated by a non-unset state, the request returns $\bot$.) For other methods, UpdateCS runs the call method of the code of the contract stored at cID on args. As an example, InitCS may populate the account balances of a set of parties as specified in the genesis block.

To ensure atomic execution, a call operates on an ephemeral deep copy CS' of the state. This emulation contains standard instructions applied to the current memory of CS'; if an instruction fails, the procedure halts and returns $\bot$, discarding CS' without mutating the master state CS. A contract may also place a call to the call' method of another contract dID on args', v'. The former is implemented in the natural way by applying UpdateCS recursively to CS' and the transaction $(\mathsf{dID}, \mathsf{call}', \mathsf{args}'; v', \mathsf{cID})$, where v' is the amount of funds transferred. Prior to the recursive call, UpdateCS checks that v' is non-negative, that the destination contract dID is initialized, and that the caller cID has sufficient funds; if so, it subtracts v' from cID and adds it to dID. On success, the returned value y' is written into the memory of the calling contract cID; on failure the recursive $\bot$ bubbles up the call stack, so all partial state changes and fund transfers are discarded. This design follows Ethereum's approach to executing transactions and transferring funds, simplified to the scope of this paper.

To enable a fully modular analysis of layer-2 protocols, our mainchain functionality allows contracts to place calls to other UC ideal functionalities and protocols. We enable this by letting $\mathcal{G}_{\mathsf{MC}}$ forward instructions to ITIs on behalf of the original party (and the session) that initiated the call, and relaying the answers back. This retains PID-wise connectivity.

APPENDIX D
DISTRIBUTED HOMOMORPHIC ENCRYPTION

LAGRANGE INTERPOLATION. Let f be a polynomial such that $\deg(f) \leq t$, $f(0) = s$, and $f(i) = s_i$ for $i \in [n]$.

Functionality 14: $\mathcal{G}_{\mathsf{MC}}[\mathbf{P}, \mathsf{Val}_{\mathsf{MC}}, \Delta_{\mathsf{lat}}, \Delta_{\mathsf{slack}}, \mathsf{Genesis}]$

Parameters
1: Hybrids: $\mathcal{G}_{\mathsf{Clock}}[\mathbf{P}']$
2: Requires: $\mathbf{P} \subseteq \mathbf{P}'$

Initialize()
3: buffer $\leftarrow [\,]$; log $\leftarrow [\,]$; eventID $\leftarrow 0$
4: **for** P $\in \mathbf{P}$ **do** $\mathsf{ptr}_{\mathsf{P}} \leftarrow 0$ // single $\mathsf{ptr}_{\mathsf{P}}$ across sid's

$\mathsf{P}_{\mathsf{sid}}$.Submit(tx) for P $\in \mathbf{P}$; if gv: any sid
5: eventID $\leftarrow$ eventID $+ 1$
6: $r \leftarrow \mathsf{T}()$
7: buffer[eventID] $\leftarrow (\mathsf{tx}, r)$
8: $\mathsf{Sim}(\mathsf{P}_{\mathsf{sid}}.\mathsf{Submit}, \mathsf{eventID}, (\mathsf{tx}, r))$
9: **return** $\perp$

$\mathsf{P}_{\mathsf{sid}}$.Read() for P $\in \mathbf{P}$; if gv: any sid
10: $(\mathbf{I}, p) \leftarrow \mathsf{Sim}(\mathsf{P}_{\mathsf{sid}}.\mathsf{Read})$
11: $r \leftarrow \mathsf{T}()$
12: $\mathbf{I} \leftarrow \mathsf{San}_{\mathbf{I}}(\mathbf{I}, r)$
13: **for** $i \in \mathbf{I}$ **do**
14: **if** buffer$[i] \neq \perp$ **then**
15: $b \leftarrow \mathsf{Val}_{\mathsf{MC}}(\log.\mathsf{tx}, \mathsf{buffer}[i].\mathsf{tx})$
16: **if** b **then**
17: log $\leftarrow$ log : (buffer$[i]$.tx, r)
18: buffer$[i] \leftarrow \perp$
19: $p \leftarrow \mathsf{San}_{\mathsf{ptr}}(p, \mathsf{ptr}_{\mathsf{P}}, r)$
20: **if** log $= [\,] \vee p = \perp$ **then return** $[\,]$
21: $\mathsf{ptr}_{\mathsf{P}} \leftarrow p$
22: **return** log[: $\mathsf{ptr}_{\mathsf{P}}$]

Proc. $\mathsf{San}_{\mathbf{I}}(\mathbf{I}, r)$ // returns an index list
23: **for** $i \in \mathbf{I}$ **do**
24: **if** buffer$[i] = \perp$ **then** $\mathbf{I} \leftarrow \mathbf{I} \setminus \{i\}$
25: **for** $i \in [i :$ buffer$[i] \neq \perp \wedge i \notin \mathbf{I}]$ **do**
26: **if** $r -$ buffer$[i].r \geq \Delta_{\mathsf{lat}}$ **then**
27: $\mathbf{I} \leftarrow \mathbf{I} : i$ // liveness: tx's get added
28: **return** $\mathbf{I}$

Proc. $\mathsf{San}_{\mathsf{ptr}}(p, p', r)$ // returns a pointer or $\perp$
29: $p^* \leftarrow \max(\{i : r - \log[i].r \geq \Delta_{\mathsf{slack}}\} \cup \{p\})$
30: $p^* \leftarrow$ **if** $p' = \perp$ **then** p^* **else** $\max(\{p^*, p'\})$
31: **if** $p^* < 0$ **then return** $\perp$
32: **if** $p^* > |\log| - 1$ **then** $p^* \leftarrow |\log| - 1$
33: **return** p^*

Proc. $\mathsf{Val}_{\mathsf{MC}}(\log, \mathsf{tx})$
34: log $\leftarrow$ log : tx
35: **return** (ContractState(Genesis, log) $\neq \perp$)

Proc. ContractState(Genesis, log)
36: CS $\leftarrow$ InitCS(Genesis)
37: **for** tx $\in$ log **do**
38: (cID, call, args; v, sID) $\leftarrow$ parse(tx)
39: **if** $v \neq 0 \vee$ sID $\neq \varepsilon$ **then return** $\perp$
// top-level isolation
40: res $\leftarrow$ UpdateCS(CS, tx)
41: **if** res $= \perp$ **then return** $\perp$
42: (CS, _) $\leftarrow$ res
43: **return** CS

Proc. InitCS(Genesis)
44: **for** (code, vk, v) $\in$ Genesis **do**
45: cID $\leftarrow$ Hash(code)
46: CS[cID] $\leftarrow$ (code, $\perp$, v)
47: **return** CS

Proc. UpdateCS(CS, tx)
48: (cID, call, args; v, sID) $\leftarrow$ parse(tx)
49: **if** (call = install) **then**
50: code $\leftarrow$ parse(args)
51: cID $\leftarrow$ Hash(code)
52: **if** CS[cID] $\neq \perp$ **then return** $\perp$
53: CS[cID] $\leftarrow$ (code, $\perp$, \$0) // mem unset
54: **return** (CS, ε)
55: **if** CS[cID] $= \perp$ **then return** $\perp$
56: CS$'$ $\leftarrow$ CS // ephemeral deep copy
57: runCS$'$[cID].code.call, updating CS$'$ as follows:
58: **on** standard instruction INS:
59: mem$'$ $\leftarrow$ INS(CS$'$[cID].mem, args; v, sID)
60: **if** mem$'$ $= \perp$ **then return** $\perp$
61: CS$'$[cID].mem $\leftarrow$ mem$'$
62: **on** dID.call$'$(args$'$; v'):
63: **if** CS$'$[dID] $= \perp$ **then return** $\perp$
64: **if** $v' < 0$ **then return** $\perp$
65: **if** CS$'$[cID].funds $- v' < 0$ **then return** $\perp$
66: CS$'$[cID].funds $\leftarrow$ CS$'$[cID].funds $- v'$
67: CS$'$[dID].funds $\leftarrow$ CS$'$[dID].funds $+ v'$
68: res $\leftarrow$ UpdateCS(CS$'$, (dID, call$'$, args$'$; v', cID))
69: **if** res $= \perp$ **then return** $\perp$
70: (CS$'$, y') $\leftarrow$ res
71: CS$'$[cID].mem.$y \leftarrow y'$
72: **on** $\mathcal{F}$ call instruction:
73: call $\mathcal{F}$ from $\mathcal{G}_{\mathsf{MC}}$ with P as originator
74: bind the response and continue
75: **on** `return`(y):
76: **return** (CS$'$, y)
77: **return** (CS$'$, ε)

Let $\lambda_{S,i}(x) := \Pi_{j \in S \setminus \{i\}}(j - x)/(j - i)$. Then $f(x) = \sum_{i \in S} \lambda_{S,i}(x) s_i$ for any $S \subseteq [n]$ with $|S| = t + 1$. We denote the operation of computing the degree-$\leq t$ polynomial f by $\mathsf{Lagrange}(t, S, [s_i : i \in S])(x)$. Given g^{s_i} for $i \in S$, one can also recover $g^{f(x)}$ by interpolation in the exponent (since $f(x)$ is a linear combination of the s_i with publicly computable coefficients $\lambda_{S,i}(x)$). We denote this procedure by $\mathsf{ExpLagrange}(t, S, [g^{s_i} : i \in S])(x)$.

We first present a distributed key generation functionality essential for realizing our distributed homomorphic encryption functionality. Func. 15 is derived from that of [9] (Fig. 1) and idealizes polynomial-based distributed key generation. In round T_1, if the corruption size of $\mathbf{KC}$ exceeds t_{KG}, the simulator provides a polynomial f of degree at most t_{KG}; otherwise, the simulator outputs the secret key shares of the corrupted parties (if fewer than t_{KG} shares are output by the simulator, the remaining are taken as 0). The secret key $\mathbf{sk}[0]$ is chosen uniformly at random by the ideal functionality, and the public key $\mathbf{pk}[0]$ is thereby fixed. A polynomial f is interpolated (using Lagrange interpolation) from the secret shares of the corrupted parties and $\mathbf{sk}[0]$.

In either case above, an honest party Q's secret key share is f evaluated at its identifier Q (viewed as an integer). The public key $\mathbf{pk}[0]$, the calling party's secret share $\mathbf{sk}[\mathsf{P}]$, and the vector of public key shares $\mathbf{pk}$ are returned.

We can UC-realize $\mathcal{F}_{\mathsf{KGen}}$ using essentially the same protocol that realizes the Katz functionality of [9] in the $\mathcal{F}_{\mathsf{PKI}}$, $\mathcal{F}_{\mathsf{CRS}}$,

Functionality 15: $\mathcal{F}_{\mathsf{KGen}}[\mathbf{KC}, G, t_{\mathsf{KG}}, \Delta_1]$

Parameters

Hybrids: $\mathcal{G}_{\mathsf{Clock}}[\mathbf{KC}']$
Requires: $\mathbf{KC} \subseteq \mathbf{KC}'$

Initialize()

$\mathsf{done} \leftarrow \bot$
$f \leftarrow \bot$
$T_0 \leftarrow \mathsf{T}()$
$T_1 \leftarrow T_0 + \Delta_1$
for $\mathsf{P} \in \mathbf{KC} \cup \{0\}$ **do**
  $(\mathbf{sk}[\mathsf{P}], \mathbf{pk}[\mathsf{P}]) \leftarrow (\bot, \bot)$

P.Setup() for $\mathsf{P} \in \mathbf{KC}$

if $\mathsf{T}() < T_0$ **then return** $\bot$
if $T_0 \leq \mathsf{T}() < T_1$ **then**
  $\mathsf{Sim}(\mathsf{P.Setup})$ // network
  return $\bot$
if $\mathsf{T}() = T_1$ **then**
  if $|\mathbf{KC} \cap \mathbf{C}| > t_{\mathsf{KG}}$ **then**
    $f' \leftarrow \mathsf{Sim}(\mathsf{P.Setup})$
    $\mathsf{San}(f')$: $\deg(f') \leq t_{\mathsf{KG}} \vee f' = \bot$
    if $\neg\mathsf{done}$ **then** $(f, \mathsf{done}) \leftarrow (f', \top)$
  if $|\mathbf{KC} \cap \mathbf{C}| \leq t_{\mathsf{KG}}$ **then**
    $\mathbf{sk}'' \leftarrow \mathsf{Sim}(\mathsf{P.Setup})$
    $\mathsf{San}(\mathbf{sk}'')$: $(\forall \mathsf{Q})\ \mathbf{sk}''[\mathsf{Q}] \in \mathbb{Z}_q \cup \{\bot\} \wedge (\mathsf{Q} \notin \mathbf{C} \Rightarrow \mathbf{sk}''[\mathsf{Q}] = \bot)$
    if $\neg\mathsf{done}$ **then** $(\mathbf{sk}', \mathsf{done}) \leftarrow (\mathbf{sk}'', \top)$
    let $\mathbf{C}'$ be $\mathbf{KC} \cap \mathbf{C}$ extended by the lowest-indexed parties of $\mathbf{KC}$ until $|\mathbf{C}'| = t_{\mathsf{KG}}$
    for $\mathsf{Q} \in \mathbf{C}'$ **do**
      $\mathbf{sk}[\mathsf{Q}] \leftarrow 0$
      if $\mathbf{sk}'[\mathsf{Q}] \neq \bot$ **then** $\mathbf{sk}[\mathsf{Q}] \leftarrow \mathbf{sk}'[\mathsf{Q}]$
    if $\mathbf{sk}[0] = \bot$ **then** $\mathbf{sk}[0] \xleftarrow{\$} \mathbb{Z}_q$
    $f \leftarrow \mathsf{Lagrange}(t_{\mathsf{KG}}, \mathbf{C}' \cup \{0\}, [\mathbf{sk}[\mathsf{Q}] : \mathsf{Q} \in \mathbf{C}' \cup \{0\}])$
  for $\mathsf{Q} \in \mathbf{KC} \cup \{0\}$ **do**
    $\mathbf{sk}[\mathsf{Q}] \leftarrow f(\mathsf{Q}); \mathbf{pk}[\mathsf{Q}] \leftarrow g^{\mathbf{sk}[\mathsf{Q}]}$
  $\mathsf{Sim}(\mathsf{P.Setup}, \mathbf{pk}[0], \mathbf{pk})$
  return $(\mathbf{pk}[0], \mathbf{pk}, \mathbf{sk}[\mathsf{P}])$
if $\mathsf{T}() > T_1$ **then return** $\bot$

and $\mathcal{F}_{\mathsf{BC}}$ hybrid model in the presence of $\mathcal{G}_{\mathsf{Clock}}$, where $\mathcal{F}_{\mathsf{BC}}$ is a Δ-delayed broadcast functionality, using a **CPA**-secure PKE and an identity-based simulation-sound NIZK-PoK. The proof is omitted to avoid repetition. Note that in our $\mathcal{F}_{\mathsf{DHE}}$ functionality, we assume $\mathcal{F}_{\mathsf{KGen}}$ functionality always outputs well-formed keys or $\bot$; thus, any realization of Func. 15 must ensure that any output key is the evaluation at 0 of a polynomial of degree at most t_{KG}.

Func. 16 presents the complete pseudo-code for $\mathcal{F}_{\mathsf{DHE}}$. We formulate $\mathcal{F}_{\mathsf{DHE}}$ in a synchronous model with respect to a shared clock functionality that provides a consistent view of the current round to all parties registered with the clock. This choice, coupled with a synchronous network resource,[5] allows us to define and realize desirable notions of *liveness*. Concretely, $\mathcal{F}_{\mathsf{DHE}}$ comes with a Setup interface that allows the $\mathbf{DC}$ to generate public parameters pp^*. Liveness of this interface ensures that as long as the $\mathbf{DC}$ is honest, a common pp^* is made available. Moreover, our definition ensures agreement on pp^* even when the $\mathbf{DC}$ is dishonest. The Encrypt interface of $\mathcal{F}_{\mathsf{DHE}}$ models a local (non-interactive) procedure to prepare votes confidentially as long as the $\mathbf{DC}$ is honest. As a result, users obtain an encrypted and verifiable vote c. Here verifiability means the vote encrypted in c can be checked to conform to a 'vote validity' predicate, specified by Val, using the VerifyEnc interface. The Eval interface of $\mathcal{F}_{\mathsf{DHE}}$ allows homomorphic tallying of a list of c's to obtain an encrypted tally CT. Here we circumvent difficulties in defining an ideal functionality for homomorphic encryption, which arise because homomorphic evaluation is a public operation, making it unclear how to define an ideal functionality that guarantees correct decryption of evaluated ciphertexts. We do this by essentially requiring that CT comes with a non-interactive zero-knowledge (NIZK) proof-of-knowledge (PoK) of correct evaluation. We then require the simulator to output the plaintexts for all ciphertexts in an evaluated CT. This is then realized via a recursive extraction procedure: first the CT extractor recovers the (verifiable) ciphertexts underlying an encrypted tally CT, then through a second layer of extraction, it can obtain the votes underlying each of those ciphertexts. The Decrypt interface models joint decryption of an evaluated ciphertext CT, producing a result res^* and publicly verifiable proofs of correct decryption π^*.[6] The functionality also provides a liveness guarantee, in the case of an honest $\mathbf{DC}$, that the output proof verifies. The decryption proofs π can be verified using the VerifyDec interface.

A technical choice in defining the above verification interfaces is whether the simulator should return a verification *bit* or a full *witness*. Recall that in the standard ideal functionality for NIZK (see Func. 9), the functionality must decide whether an input statement x is valid. This is defined by requiring the simulator to output a witness w and then using the relation $\mathsf{R}(x, w)$ underlying the NIZK to decide the validity of x. In our context, a bit formulation is meaningful (and realizable) since the relations corresponding to well-formedness of ciphertexts and decryption results are essentially implemented through tables internal to the functionality.

Protocol 17 presents the complete pseudo-code for π_{DHE}. While $\mathcal{F}_{\mathsf{KGen}}$ is used to realize the Setup interface of $\mathcal{F}_{\mathsf{DHE}}$, we implement the Encrypt interface of $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}$ via ElGamal encryption augmented with NIZKs of vote correctness. The Eval interface exploits the homomorphic property of ElGamal (for small votes). It checks all the proofs for all input ciphertexts,

[5] Synchrony, in practice, is a reasonable assumption as an adversary would have to corrupt/control a large number of network routers to disrupt it.

[6] In defining this functionality, our assumption is that in realizations a special error message is returned whenever a result-proof pair does not verify.

Functionality 16: $\mathcal{F}_{\mathsf{DHE}}[\mathbf{P}, \mathbf{DC}, t_{\mathsf{DC}}, \mathsf{Val}, F, \Delta_1, \Delta_2, \Delta_3, G]$

Parameters

Hybrids: $\mathcal{G}_{\mathsf{Clock}}[\mathbf{P}']$
Requires: $\mathbf{DC} \cap \mathbf{P} = \varnothing \wedge \mathbf{P} \cup \mathbf{DC} \subseteq \mathbf{P}'$

Initialize()

$\mathsf{done}_1, \mathsf{done}_2, \mathsf{pp}^*, \mathsf{CT}^*, \mathsf{M}^*, b^*, \mathsf{res}^*, \pi^* \leftarrow \bot$
for $\mathsf{P} \in \mathbf{P}$: $\mathsf{in}[\mathsf{P}] \leftarrow \bot$
$\mathsf{Dec}[*,*], \mathsf{VerEnc}[*,*], \mathsf{Ver}[*,*,*,*], \mathsf{Res}[*,*] \leftarrow \bot$
$T_0 \leftarrow \mathsf{T}()$
$T_1 \leftarrow T_0 + \Delta_1; T_2 \leftarrow T_1 + \Delta_2$; $T_3 \leftarrow T_2 + \Delta_3$

P.Setup() for $\mathsf{P} \in \mathbf{DC}$

if $\mathsf{T}() < T_0$ **then return** $\bot$
if $T_0 \leq \mathsf{T}() < T_1$ **then**
  Sim(P.Setup)
  return $\bot$
if $\mathsf{T}() \geq T_1$ **then**
  $\mathsf{pp}' \leftarrow$ Sim(P.Setup)
  if $\neg \mathsf{done}_1$ **then** $(\mathsf{pp}^*, \mathsf{done}_1) \leftarrow (\mathsf{pp}', \top)$
  if $|\mathbf{DC} \cap \mathbf{C}| \leq t_{\mathsf{DC}}$ **then**
    San(pp^*): $\mathsf{pp}^* \neq \bot$
  return pp^*

P.Encrypt(pp, m) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{DC}$

check $\mathsf{Val}(m)$
if $\mathsf{pp} = \mathsf{pp}^* \wedge |\mathbf{DC} \cap \mathbf{C}| \leq t_{\mathsf{DC}}$ **then**
  $c \leftarrow$ Sim(P.Encrypt, pp, $1^{|m|}$)
if $\mathsf{pp} = \mathsf{pp}^* \wedge |\mathbf{DC} \cap \mathbf{C}| > t_{\mathsf{DC}}$ **then**
  $c \leftarrow$ Sim(P.Encrypt, pp, m)
if $\mathsf{pp} \neq \mathsf{pp}^*$ **then**
  $c \leftarrow$ Sim(P.Encrypt, pp, m)
San(c): $c \neq \bot \wedge \mathsf{Dec}[\mathsf{pp}, c] = \bot$
$\mathsf{Dec}[\mathsf{pp}, c] \leftarrow m$
$\mathsf{VerEnc}[\mathsf{pp}, c] \leftarrow 1$
return c

P.VerifyEnc(pp, c) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{DC}$; if gv: any sid

$b \leftarrow$ Sim(P.VerifyEnc, pp, c)
San(b): $b \in \{0, 1\}$
if $\mathsf{VerEnc}[\mathsf{pp}, c] \neq \bot$ **then**
  return $\mathsf{VerEnc}[\mathsf{pp}, c]$
if $\mathsf{Dec}[\mathsf{pp}, c] \neq \bot$ **then**
  return 1
$\mathsf{VerEnc}[\mathsf{pp}, c] \leftarrow b$
return $\mathsf{VerEnc}[\mathsf{pp}, c]$

P.Eval(pp, B) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{DC}$; if gv: any sid

if $\exists (c \in B)\ \mathsf{VerEnc}[\mathsf{pp}, c] = 0$ **then return** $\bot$
$(\mathsf{CT}, \mathsf{M}) \leftarrow$ Sim(P.Eval, pp, B)
for $c \in B$ with $\mathsf{Dec}[\mathsf{pp}, c] = \bot \wedge \mathsf{Val}(\mathsf{M}[c])$ **do**
  $\mathsf{Dec}[\mathsf{pp}, c] \leftarrow \mathsf{M}[c]$
  $\mathsf{VerEnc}[\mathsf{pp}, c] \leftarrow 1$
for $c \in B$ with $\mathsf{Dec}[\mathsf{pp}, c] = \bot$ **do**
  $\mathsf{VerEnc}[\mathsf{pp}, c] \leftarrow 0$
if $(\exists c \in B)\mathsf{Dec}[\mathsf{pp}, c] = \bot$ **then return** $\bot$
$\mathsf{res} \leftarrow F([\mathsf{Dec}[\mathsf{pp}, c] : c \in B])$
San(CT): $\mathsf{CT} \neq \bot \wedge (\mathsf{Res}[\mathsf{pp}, \mathsf{CT}] = \bot \vee \mathsf{Res}[\mathsf{pp}, \mathsf{CT}] = \mathsf{res})$
$\mathsf{Res}[\mathsf{pp}, \mathsf{CT}] \leftarrow \mathsf{res}$
return CT

P.Decrypt(CT) for $\mathsf{P} \in \mathbf{DC}$, once

if $\mathsf{T}() < T_2$ **then return** $\bot$
if $T_2 \leq \mathsf{T}() < T_3$ **then**
  Sim(P.Decrypt, CT)
  if $\mathsf{in}[\mathsf{P}] = \bot$: $\mathsf{in}[\mathsf{P}] \leftarrow \mathsf{CT}$
  return $\bot$
if $\mathsf{T}() = T_3$ **then**
  $(\mathsf{CT}', \mathsf{M}', \pi') \leftarrow$ Sim(P.Decrypt, CT)
  if $\neg \mathsf{done}_2$ **then** $(\mathsf{CT}^*, \mathsf{M}^*, \pi^*, \mathsf{done}_2) \leftarrow (\mathsf{CT}', \mathsf{M}', \pi', \top)$
  if $(\exists \mathsf{CT}'')|\{\mathsf{P} \in \mathbf{DC} \setminus \mathbf{C} : \mathsf{in}[\mathsf{P}] = \mathsf{CT}''\}| \geq |\mathbf{DC}| - t_{\mathsf{DC}}$ **then**
    $\mathsf{CT}^* \leftarrow \mathsf{CT}''$
  if $\mathsf{CT}^* \neq \mathsf{in}[\mathsf{P}]$: **return** $\bot$
  if $\mathsf{Res}[\mathsf{pp}^*, \mathsf{CT}^*] \neq \bot$ **then**
    $\mathsf{res}^* \leftarrow \mathsf{Res}[\mathsf{pp}^*, \mathsf{CT}^*]$
  else
    $\mathsf{Rng} \leftarrow [j : \mathsf{M}^*[j]$ defined$]$
    for $j \in \mathsf{Rng}$ with $\mathsf{M}^*[j] =$ "$\mathrm{dec}(c)$" **do**
      $\mathsf{M}^*[j] \leftarrow \mathsf{Dec}[\mathsf{pp}^*, c]$
    if $(\exists j \in \mathsf{Rng})\neg\mathsf{Val}(\mathsf{M}^*[j])$ **then return** $\bot$
    $\mathsf{res}^* \leftarrow F([\mathsf{M}^*[j] : j \in \mathsf{Rng}])$
    $\mathsf{Res}[\mathsf{pp}^*, \mathsf{CT}^*] \leftarrow \mathsf{res}^*$
  if $|\mathbf{DC} \cap \mathbf{C}| \leq t_{\mathsf{DC}}$ **then**
    San(π^*): $\pi^* \neq \bot \wedge \mathsf{Ver}[\mathsf{pp}^*, \mathsf{CT}^*, \mathsf{res}^*, \pi^*] \neq 0$
    $\mathsf{Ver}[\mathsf{pp}^*, \mathsf{CT}^*, \mathsf{res}^*, \pi^*] \leftarrow 1$
  if $\pi^* \neq \bot \wedge \mathsf{Ver}[\mathsf{pp}^*, \mathsf{CT}^*, \mathsf{res}^*, \pi^*] \in \{1, \bot\}$ **then**
    $\mathsf{Ver}[\mathsf{pp}^*, \mathsf{CT}^*, \mathsf{res}^*, \pi^*] \leftarrow 1$
    return (res^*, π^*)
  else
    return $\bot$
if $\mathsf{T}() > T_3$ **then return** $\bot$

P.VerifyDec(pp, CT, res, π) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{DC}$; if gv: any sid

$b \leftarrow$ Sim(P.VerifyDec, pp, CT, res, π)
San(b): $b \in \{0, 1\}$
if $\mathsf{Ver}[\mathsf{pp}, \mathsf{CT}, \mathsf{res}, \pi] \neq \bot$ **then**
  return $\mathsf{Ver}[\mathsf{pp}, \mathsf{CT}, \mathsf{res}, \pi]$
if $\mathsf{Res}[\mathsf{pp}, \mathsf{CT}] \neq \bot$ **then**
  if $\mathsf{Res}[\mathsf{pp}, \mathsf{CT}] \neq \mathsf{res}$ **then**
    $\mathsf{Ver}[\mathsf{pp}, \mathsf{CT}, \mathsf{res}, \pi] \leftarrow 0$
    return $\mathsf{Ver}[\mathsf{pp}, \mathsf{CT}, \mathsf{res}, \pi]$
  if $\mathsf{Res}[\mathsf{pp}, \mathsf{CT}] = \mathsf{res}$ **then**
    $\mathsf{Ver}[\mathsf{pp}, \mathsf{CT}, \mathsf{res}, \pi] \leftarrow b$
    return $\mathsf{Ver}[\mathsf{pp}, \mathsf{CT}, \mathsf{res}, \pi]$
$\mathsf{Ver}[\mathsf{pp}, \mathsf{CT}, \mathsf{res}, \pi] \leftarrow b$
if $b = 1$ **then**
  $\mathsf{Res}[\mathsf{pp}, \mathsf{CT}] \leftarrow \mathsf{res}$
return $\mathsf{Ver}[\mathsf{pp}, \mathsf{CT}, \mathsf{res}, \pi]$

aborting if any fails, and then computes the product of the input ciphertexts, together with a NIZK proof that (1) all input proofs verified and (2) the evaluated ciphertext was correctly multiplied. This proof needs to be succinct, as the input set of ciphertexts can be large. The decryption interface Decrypt, run by members of the $\mathbf{DC}$, parses CT as $(\mathsf{CT}_1, \mathsf{CT}_2, \pi_{\mathsf{agg}})$ and verifies the evaluation proof π_{agg} for the encrypted tally $(\mathsf{CT}_1, \mathsf{CT}_2)$. It then retrieves P's secret key $\mathsf{sk}[\mathsf{P}]$ and partial public key y_{P} computed during Setup and computes its share of ElGamal decryption $d_{\mathsf{P}} = \mathsf{CT}_1^{\mathsf{sk}[\mathsf{P}]}$. Each party then attaches a proof π_{P}, using witness $\mathsf{sk}[\mathsf{P}]$, that $(g, \mathsf{CT}_1, y_{\mathsf{P}}, d_{\mathsf{P}})$ is a valid DDH tuple (where g is the group generator). Each party broadcasts its decryption share and its proof to all other parties. Next, parties receive the shares and proofs and collect the shares with verified proofs in Shares. Finally, each party runs polynomial interpolation using the valid decryption shares at 0 to compute $\mathsf{CT}_1^{\mathsf{sk}[0]}$. This is used to decrypt the ElGamal ciphertext as usual, to compute the result tally. The result and

the collected proofs are returned as the final output. VerifyDec is similar in that it collects the shares with verified proofs to recompute the result. It then checks whether the computed result matches the input tally. Note that in Protocol 17, we abbreviate the functionality references for compactness by not mentioning the complete parameters.

A. Security of π_{DHE}

Theorem 3 (Theorem 1, formal version)**.** *Let* $\mathbf{P}$, $\mathbf{DC}$, t_{DC}, Val, F, Δ, Δ_1, Δ_2, Δ_3, G *be a given set of parameters with* $\mathbf{DC}$ *a set of parties of size* n *and* $t_{\mathsf{DC}} < \lfloor (n+1)/2 \rfloor$, $\mathsf{Val}(m) := (m \in [0, b])$ *for some integer* b*, and* $F(M) := \sum_{m \in M} m$*. Let* π_{DHE} *be as described in Prot. 17 for this set of parameters and with respect to hybrids* $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}]$, $\mathcal{F}_{\mathsf{NIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{DDH}}]$, $\mathcal{F}_{\mathsf{iNIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{F}_{\mathcal{R}_{\mathsf{AGG}}}]$, $\mathcal{F}_{\mathsf{KGen}}[\mathbf{DC}, G, t_{\mathsf{DC}}, \Delta_1]$, n *sessions of* $\mathcal{F}_{\mathsf{BC}}[\mathsf{P}, \mathbf{DC}, \Delta/2]$ *for* $\mathsf{P} \in \mathbf{DC}$, $\mathcal{G}_{\mathsf{Clock}}[\mathbf{P} \cup \mathbf{DC}]$*, and a group scheme* G *where the DDH problem is hard and which outputs groups of prime order* p *(and thus ElGamal encryption in the exponent is CPA-secure). Suppose further that all delay parameters are positive and* $b \ll p$ *so that computing the discrete log of* g^m *for a valid* m *is easy. Then* π_{DHE} *UC-realizes* $\mathcal{F}_{\mathsf{DHE}}$ *with parameters* $\mathbf{P}$, $\mathbf{DC}$, t_{DC}, Val, F, Δ, Δ_1, Δ_2, $\Delta_3 + \Delta/2$, G.

Proof. For any real-world adversary $\mathcal{A}$, we give a simulator Sim such that no environment can distinguish between $(\pi_{\mathsf{DHE}}, \mathcal{A})$ and $(\mathcal{F}_{\mathsf{DHE}}, \mathsf{Sim})$. The simulator internally runs hybrids $\mathcal{F}_{\mathsf{BC}}$, $\mathcal{F}_{\mathsf{KGen}}$, $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathcal{R}_{\mathsf{VOTE}}]$, $\mathcal{F}_{\mathsf{NIZK}}[\mathcal{R}_{\mathsf{DDH}}]$, and $\mathcal{F}_{\mathsf{iNIZK}}[\mathcal{F}_{\mathcal{R}_{\mathsf{AGG}}}]$. Calls to the simulator in these hybrids are forwarded to $\mathcal{A}$. Further, as mentioned in Section VII, the simulator also responds to invocations of the $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathcal{R}_{\mathsf{VOTE}}]$.P.Verify interface by the environment. Note that since $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathcal{R}_{\mathsf{VOTE}}]$ is a globally verifiable local functionality, the environment can only invoke the Verify interface, albeit from any session. The simulator must provide verification responses (only on the proofs from the unique session that it simulates), which are processed by the ideal functionality code to produce the verification bit that is then returned to the environment. This is the key difference between a local functionality (where the environment cannot access any of the interfaces) and a globally verifiable local functionality. We show later how the simulator gives consistent responses to these queries based on its internal simulation of the $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathcal{R}_{\mathsf{VOTE}}]$ hybrid.

On being invoked in any of the interfaces listed below, the simulator will essentially follow the specification of π_{DHE} except when it doesn't have the required 'secrets', or it is required to output something 'extra' over what is computed in π_{DHE}. It will faithfully simulate all the hybrids internally, including independently maintaining lists to mimic the ideal functionality lists. This helps it adapt its responses to match the real-world responses, even if the adversary provides an incorrect or inconsistent response. As we will argue, this also renders the code for output sanitization or consistency checks in several interfaces of $\mathcal{F}_{\mathsf{DHE}}$ essentially redundant, since the outputs are already sanitized within the simulated hybrids. We argue that the ideal-world execution $(\mathcal{F}_{\mathsf{DHE}}, \mathsf{Sim})$ (the combined code of the ideal functionality and the described simulator) is indistinguishable from that of $(\pi_{\mathsf{DHE}}, \mathcal{A})$.

We now describe the simulator Sim with respect to parameters $\mathbf{P}, \mathbf{DC}, t_{\mathsf{DC}}, \mathsf{Val}, F, \Delta, \Delta_1, \Delta_2, \Delta_3$, and G.

$\mathcal{F}_{\mathsf{DHE}}$.P.Setup:

Sim(P.Setup): The simulator internally simulates $\mathcal{F}_{\mathsf{KGen}}[\mathbf{DC}, G, t_{\mathsf{DC}}, \Delta_1]$. Note that the call to the simulator of $\mathcal{F}_{\mathsf{KGen}}$ is passed to the adversary $\mathcal{A}$. The simulator returns $\perp$.

$\mathsf{pp}' \leftarrow$ Sim(P.Setup): The simulator calls the adversary in $\mathcal{F}_{\mathsf{KGen}}$ to receive f' or $\mathbf{sk}''$ depending on whether $|\mathbf{DC} \cap \mathbf{C}| > t_{\mathsf{DC}}$ or not. It finishes the simulation of $\mathcal{F}_{\mathsf{KGen}}$.Setup to generate $(\mathsf{pk}, [y_j : j \in [n]], \mathsf{sk}[\mathsf{P}])$. It stores $\mathsf{sk}[\mathsf{P}]$, sets $\mathsf{pp}' = (\mathsf{pk}, [y_j : j \in [n]])$, and returns it. Note that the simulation of $\mathcal{F}_{\mathsf{KGen}}$ ensures that the public parameters and secret keys of all participants are generated once, and that they are not $\perp$ in the honest-majority case ($|\mathbf{DC} \cap \mathbf{C}| \leq t_{\mathsf{DC}}$, $t_{\mathsf{DC}} < \lfloor (n+1)/2 \rfloor$); thus, code lines 15-16 of $\mathcal{F}_{\mathsf{DHE}}$ do not affect pp'. Thus, the code of $\mathcal{F}_{\mathsf{DHE}}$ and π_{DHE} is identical for the Setup interface.

$\mathcal{F}_{\mathsf{DHE}}$.P.Encrypt:

$c \leftarrow$ Sim(P.Encrypt, $\mathsf{pp}^*, 1^{|m|}$): If $\mathsf{ppCheck}(\mathsf{pp}^*)$, then the simulator chooses an arbitrary message msg of length $|m|$ satisfying the Val predicate. It encrypts msg using ElGamal encryption with uniformly chosen randomness r to generate the ciphertext c, and then calls $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}]$.P.Prove$((\mathsf{pk}, c), (\mathsf{msg}, r))$ to get a proof π_{vote}. As mentioned earlier, it also updates an internal $\mathsf{Ver}_{\mathsf{int}}$ list to mimic the $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}]$'s Ver list and sets $\mathsf{Ver}_{\mathsf{int}}[(\mathsf{pk}, c), \pi_{\mathsf{vote}}]$ to 1. Additionally it stores (msg, r), the encryption witness in another local list $\mathsf{Wit}_{\mathsf{int}}[(\mathsf{pk}, c), \pi_{\mathsf{vote}}] = (\mathsf{msg}, r)$. This list will be used to return consistent responses to the verification queries issued by the environment. It maintains its own $\mathsf{Dec}_{\mathsf{int}}$ and $\mathsf{VerEnc}_{\mathsf{int}}$ lists, and updates them as $\mathsf{Dec}_{\mathsf{int}}[\mathsf{pp}^*, c] =$ "$\mathrm{dec}(c)$" and $\mathsf{VerEnc}_{\mathsf{int}}[\mathsf{pp}^*, c] = 1$ respectively, where "$\mathrm{dec}(c)$" denotes that the plaintext corresponding to this ciphertext is the one set by the ideal functionality (line 26 in functionality $\mathcal{F}_{\mathsf{DHE}}$). It returns (c, π_{vote}). Note as c is produced using ElGamal encryption with a uniformly random r, it is not $\perp$, and collisions in c occur with low probability. Therefore the San code in $\mathcal{F}_{\mathsf{DHE}}$ (line 25) doesn't result in any change in c whp. Further note, as the plaintext for c is the one that the functionality records in its Dec list, this stored value is the one that is available in interfaces like Eval and Decrypt. Therefore, from the environment's point of view, as a result of encrypting msg, there is no change in the output of any other interface as they continue to return m as the plaintext corresponding to c. The only difference is that the environment gets an encryption of msg here instead of an encryption of m as in the real world.

$c \leftarrow$ Sim(P.Encrypt, pp^*, m): If $\mathsf{ppCheck}(\mathsf{pp}^*)$,

Protocol 17: $\pi_{\mathsf{DHE}}[\mathbf{P}, \mathbf{DC}, t_{\mathsf{DC}}, \mathsf{Val}, F, \Delta_1, \Delta_2, \Delta_3, G]$

Parameters

Hybrids: $\mathcal{G}_{\mathsf{Clock}}[\mathbf{P} \cup \mathbf{DC}]$, $\mathcal{F}_{\mathsf{iNIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{F}_{\mathcal{R}_{\mathsf{AGG}}}]$, $\mathcal{F}_{\mathsf{NIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{DDH}}]$, $\mathcal{F}^{\mathsf{gv}}_{\mathsf{NIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}]$, $\mathcal{F}_{\mathsf{BC}}[\mathsf{P}, \mathbf{DC}, \Delta/2]$ for each $\mathsf{P} \in \mathbf{DC}$, $\mathcal{F}_{\mathsf{KGen}}[\mathbf{DC}, G, t_{\mathsf{DC}}, \Delta_1]$

Requires: $\mathbf{DC} \cap \mathbf{P} = \varnothing \wedge \mathbf{P} \cup \mathbf{DC} \subseteq \mathbf{P}'$

Initialize()

$T_0 \leftarrow \mathsf{T}()$

$T_1 \leftarrow T_0 + \Delta_1$; $T_2 \leftarrow T_1 + \Delta_2$; $T_3 \leftarrow T_2 + \Delta_3$

P.Setup() for $\mathsf{P} \in \mathbf{DC}$

if $\mathsf{T}() < T_0$ **then return** $\perp$

if $T_0 \leq \mathsf{T}() < T_1$ **then**

  $\perp \leftarrow \mathcal{F}_{\mathsf{KGen}}[\mathbf{DC}, G, t_{\mathsf{DC}}, \Delta_1]$.P.Setup()

  return $\perp$

if $\mathsf{T}() \geq T_1$ **then**

  $(\mathsf{pk}, [y_j : j \in [n]], \mathsf{sk}[\mathsf{P}]) \leftarrow \mathcal{F}_{\mathsf{KGen}}$.P.Setup()

  store $\mathsf{sk}[\mathsf{P}]$

  $\mathsf{pp}^* \leftarrow (\mathsf{pk}, [y_j : j \in [n]])$

  return pp^*

P.Encrypt(pp, m) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{DC}$

if $\neg$ppCheck(pp) **then return** $\perp$

$(\mathsf{pk}, [y_j : j \in [n]]) \leftarrow \mathsf{parse}(\mathsf{pp})$

$r \xleftarrow{\$} \mathbb{Z}_q$; $(c_1, c_2) \leftarrow (g^r, g^m \cdot \mathsf{pk}^r)$

$\pi_{\mathsf{vote}} \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{NIZK}}[\mathcal{R}_{\mathsf{VOTE}}]$.P.Prove($(\mathsf{pk}, c_1, c_2), (m, r)$)

return $(c_1, c_2, \pi_{\mathsf{vote}})$

P.VerifyEnc(pp, c) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{DC}$; if gv: any sid

if $\neg$ppCheck(pp) **then return** 0

$(\mathsf{pk}, [y_j : j \in [n]]) \leftarrow \mathsf{parse}(\mathsf{pp})$

$(c_1, c_2, \pi_{\mathsf{vote}}) \leftarrow \mathsf{parse}(c)$

return $\mathcal{F}^{\mathsf{gv}}_{\mathsf{NIZK}}[\mathcal{R}_{\mathsf{VOTE}}]$.P.Verify($(\mathsf{pk}, c_1, c_2), \pi_{\mathsf{vote}}$)

P.Eval(pp, B) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{DC}$; if gv: any sid

if $\neg$ppCheck(pp) **then return** $\perp$

$(\mathsf{pk}, [y_j : j \in [n]]) \leftarrow \mathsf{parse}(\mathsf{pp})$

$\mathsf{CT}_1 \leftarrow \mathsf{CT}_2 \leftarrow 1_G$

for $(c_1, c_2, \pi_{\mathsf{vote}}) \in B$ **do**

  if $\mathcal{F}^{\mathsf{gv}}_{\mathsf{NIZK}}[\mathcal{R}_{\mathsf{VOTE}}]$.P.Verify($(\mathsf{pk}, c_1, c_2), \pi_{\mathsf{vote}}$) **then**

    $(\mathsf{CT}_1, \mathsf{CT}_2) \leftarrow (\mathsf{CT}_1 \cdot c_1, \mathsf{CT}_2 \cdot c_2)$

  else

    return $\perp$

$\pi_{\mathsf{agg}} \leftarrow \mathcal{F}_{\mathsf{iNIZK}}[\mathcal{R}_{\mathsf{AGG}}]$.P.Prove($(\mathsf{pk}, \mathsf{CT}_1, \mathsf{CT}_2), B$)

$\mathsf{CT} \leftarrow (\mathsf{CT}_1, \mathsf{CT}_2, \pi_{\mathsf{agg}})$

return CT

Proc. ppCheck(pp)

$(\mathsf{pk}, [y_j : j \in [n]]) \leftarrow \mathsf{parse}(\mathsf{pp})$

check ExpLagrange($t_{\mathsf{DC}}, [1, t_{\mathsf{DC}} + 1], [y_j : j \in [1, t_{\mathsf{DC}} + 1]]$)$(0) = \mathsf{pk}$

for $k \in [t_{\mathsf{DC}} + 2, n]$ **do**

  check ExpLagrange($t_{\mathsf{DC}}, [1, t_{\mathsf{DC}} + 1], [y_j : j \in [1, t_{\mathsf{DC}} + 1]]$)$(k) = y_k$

return 1

P.Decrypt(CT) for $\mathsf{P} \in \mathbf{DC}$, once

if $\mathsf{T}() < T_2$ **then return** $\perp$

if $\mathsf{T}() = T_2$ **then**

  if $\neg$ppCheck(pp^*) **then return** $\perp$

  if $\mathsf{pk} = \perp$ **then return** $\perp$

  if $\mathsf{tally} \neq \perp$ **then return** tally // if already computed

  $(\mathsf{CT}_1, \mathsf{CT}_2, \pi_{\mathsf{agg}}) \leftarrow \mathsf{parse}(\mathsf{CT})$

  if $\neg\mathcal{F}_{\mathsf{iNIZK}}[\mathcal{R}_{\mathsf{AGG}}]$.P.Verify($(\mathsf{pk}, \mathsf{CT}_1, \mathsf{CT}_2), \pi_{\mathsf{agg}}$) **then return** $\perp$

  $d_{\mathsf{P}} \leftarrow (\mathsf{CT}_1)^{\mathsf{sk}[\mathsf{P}]}$

  $\pi_{\mathsf{P}} \leftarrow \mathcal{F}_{\mathsf{NIZK}}[\mathcal{R}_{\mathsf{DDH}}]$.P.Prove($(g, \mathsf{CT}_1, y_{\mathsf{P}}, d_{\mathsf{P}}), \mathsf{sk}[\mathsf{P}]$)

  $\mathcal{F}_{\mathsf{BC}}[\mathsf{P}, \mathbf{DC}, \Delta/2]$.P.Cast($\mathsf{CT}, d_{\mathsf{P}}, \pi_{\mathsf{P}}$)

  return ok

if $T_2 + 1 \leq \mathsf{T}() < T_2 + \Delta/2$ **then**

  return ok

if $T_2 + \Delta/2 \leq \mathsf{T}() < T_3$ **then**

  $S \leftarrow \varnothing$; Shares $\leftarrow [\,]$

  for $j \in [n]$ **do**

    $(\mathsf{CT}^{(j)}, d_j, \pi_j) \leftarrow \mathcal{F}_{\mathsf{BC}}[\mathsf{P}_j, \mathbf{DC}, \Delta/2]$.P.Fetch()

  if $\mathsf{Maj}([\mathsf{CT}^{(j)} : j \in [n]]) \neq \mathsf{CT}$ **then return** $\perp$

  for $j \in [n]$ **do**

    if $\mathcal{F}_{\mathsf{NIZK}}[\mathcal{R}_{\mathsf{DDH}}]$.P.Verify($(g, \mathsf{CT}_1^{(j)}, y_j, d_j), \pi_j$) **then**

      Shares $\leftarrow$ Shares : d_j

      $S \leftarrow S \cup \{j\}$

      $\mathsf{proof}_j \leftarrow (d_j, \pi_j)$

    else

      $\mathsf{proof}_j \leftarrow \perp$

  $\mathsf{D} \leftarrow \mathsf{ExpLagrange}(t_{\mathsf{DC}}, S, \mathsf{Shares})(0)$

  if $\mathsf{D} = \perp$ **then return** $\perp$

  $\mathsf{tally} \leftarrow \mathsf{DL}(g, \mathsf{CT}_2/\mathsf{D})$

  $\mathsf{proof} \leftarrow [\mathsf{proof}_j : j \in [n]]$

  return ok

if $\mathsf{T}() = T_3$ **then**

  return (tally, proof)

if $\mathsf{T}() > T_3$ **then return** $\perp$

P.VerifyDec($\mathsf{pp}, \mathsf{CT}, \mathsf{tally}, \mathsf{proof}$) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{DC}$; if gv: any sid

if $\neg$ppCheck(pp) **then return** 0

$(\mathsf{pk}, [y_j : j \in [n]]) \leftarrow \mathsf{parse}(\mathsf{pp})$

$(\mathsf{CT}_1, \mathsf{CT}_2, \pi_{\mathsf{agg}}) \leftarrow \mathsf{parse}(\mathsf{CT})$

$[\mathsf{proof}_j : j \in [n]] \leftarrow \mathsf{parse}(\mathsf{proof})$ // $\perp$ if sizes don't match

if $\neg\mathcal{F}_{\mathsf{iNIZK}}[\mathcal{R}_{\mathsf{AGG}}]$.P.Verify($(\mathsf{pk}, \mathsf{CT}), \pi_{\mathsf{agg}}$) **then return** 0

$S \leftarrow \varnothing$; Shares $\leftarrow [\,]$

for $j \in [n]$ **do**

  if $\mathsf{proof}_j \neq \perp$ **then** $(d_j, \pi_j) \leftarrow \mathsf{parse}(\mathsf{proof}_j)$

  if $\mathcal{F}_{\mathsf{NIZK}}[\mathcal{R}_{\mathsf{DDH}}]$.P.Verify($(g, \mathsf{CT}_1, y_j, d_j), \pi_j$) **then**

    Shares $\leftarrow$ Shares : d_j

    $S \leftarrow S \cup \{j\}$

$\mathsf{D} \leftarrow \mathsf{ExpLagrange}(t_{\mathsf{DC}}, S, \mathsf{Shares})(0)$

if $\mathsf{D} = \perp$ **then return** 0

return $(\mathsf{tally} = \mathsf{DL}(g, \mathsf{CT}_2/\mathsf{D}))$

Relations:

$\mathcal{R}_{\mathsf{VOTE}}((y, c_1, c_2), (m, r)) := \big(c_1 = g^r \wedge c_2 = g^m \cdot y^r \wedge \mathsf{Val}(m)\big)$

$\mathcal{R}_{\mathsf{AGG}}((y, C_1, C_2), B) := \big((C_1, C_2) = (\prod_{c \in B} c.c_1, \prod_{c \in B} c.c_2) \wedge \forall c \in B : \mathcal{F}^{\mathsf{gv}}_{\mathsf{NIZK}}[\mathcal{R}_{\mathsf{VOTE}}]$.P.Verify$((y, c.c_1, c.c_2), c.\pi_{\mathsf{vote}})\big)$

$\mathcal{R}_{\mathsf{DDH}}((g, h, g', h'), x) := \big(g' = g^x \wedge h' = h^x\big)$

then the simulator encrypts m using ElGamal encryption with uniformly chosen randomness r to generate the ciphertext c, and then calls $\mathcal{F}^{\mathsf{gv}}_{\mathsf{NIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}]$.P.Prove$((\mathsf{pk}, c), (m, r))$ to get a proof π_{vote}. It updates all internal lists, $\mathsf{Dec}_{\mathsf{int}}[\mathsf{pp}^*, c] = m$ and $\mathsf{VerEnc}_{\mathsf{int}}[\mathsf{pp}^*, c] = 1$ respectively. It returns (c, π_{vote}). Note as c is produced using ElGamal encryption with a uniformly random r, it is not $\perp$, and

collisions in c occur with low probability. Therefore the San code in $\mathcal{F}_{\mathsf{DHE}}$ (line 25) doesn't result in any change in c whp.

$c \leftarrow \mathsf{Sim}(\mathsf{P.Encrypt}, \mathsf{pp}, m)$: If $\mathsf{ppCheck}(\mathsf{pp})$, then the simulator encrypts m using ElGamal encryption with uniformly chosen randomness r to generate the ciphertext c, and then calls $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}].\mathsf{P.Prove}((\mathsf{pk}, c), (m, r))$ to get a proof π_{vote} where $(\mathsf{pk}, [y_j : j \in [n]]) \leftarrow \mathsf{parse}(\mathsf{pp})$. It updates all internal lists, $\mathsf{Dec}_{\mathsf{int}}[\mathsf{pp}, c] = m$ and $\mathsf{VerEnc}_{\mathsf{int}}[\mathsf{pp}, c] = 1$ respectively. It returns (c, π_{vote}). Note as c is produced using the ElGamal encryption algorithm with a uniformly random r, it is not $\perp$, and collisions in c occur with low probability. Therefore the San code in $\mathcal{F}_{\mathsf{DHE}}$ (line 25) doesn't result in any change in c whp.

The code of $\mathcal{F}_{\mathsf{DHE}}$ and π_{DHE} for the Encrypt interface is indistinguishable. The environment's view in the ideal world though is indistinguishable as long as the encryption of an arbitrary but valid message (msg) is indistinguishable from that of a random message m. We show formally in Lemma 1 that this indistinguishability holds due to the IND-CPA security of ElGamal encryption.

$\mathcal{F}_{\mathsf{DHE}}$.P.VerifyEnc:

$b \leftarrow \mathsf{Sim}(\mathsf{P.VerifyEnc}, \mathsf{pp}, c)$: If $\mathsf{ppCheck}(\mathsf{pp})$, then the simulator parses pp to get $(\mathsf{pk}, [y_j : j \in [n]])$, and parses c to get $(c_1, c_2, \pi_{\mathsf{vote}})$. The simulator calls the adversary on $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}].\mathsf{P.Verify}((\mathsf{pk}, c_1, c_2), \pi_{\mathsf{vote}})$ to receive the witness (m, r). It finishes the simulation of $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}]$. Note, here if the adversary provides an incorrect witness for a 'proven' statement, the witness is ignored and the response is returned as per the stored response in $\mathsf{Ver}_{\mathsf{int}}[(\mathsf{pk}, c_1, c_2), \pi_{\mathsf{vote}}]$. It updates its $\mathsf{VerEnc}_{\mathsf{int}}[\mathsf{pp}, c] = b$ and $\mathsf{Dec}_{\mathsf{int}}[\mathsf{pp}, c] = m$ lists. If the interface was invoked earlier for the same message, the simulator always returns the stored response from its $\mathsf{VerEnc}_{\mathsf{int}}$ list. Similarly if its $\mathsf{Dec}_{\mathsf{int}}$ list is already set for this input, the simulator returns 1. Thus lines 30-34 from the $\mathcal{F}_{\mathsf{DHE}}$ code do not have any impact on the returned bit b. Thus the code of $\mathcal{F}_{\mathsf{DHE}}$ and π_{DHE} for the VerifyEnc interface is essentially identical given this simulator.

$\mathcal{F}_{\mathsf{DHE}}$.P.Eval:

$(\mathsf{CT}, M) \leftarrow \mathsf{Sim}(\mathsf{P.Eval}, \mathsf{pp}, B)$: If $\mathsf{ppCheck}(\mathsf{pp})$, then the simulator parses pp to get $(\mathsf{pk}, [y_j : j \in [n]])$. Here the simulator has to output the list of messages M underlying the list B, in addition to the evaluated ciphertext CT. The simulator receives m for each $c = (c_1, c_2, \pi_{\mathsf{vote}}) \in B$ from the adversary using $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}].\mathsf{P.Verify}((\mathsf{pk}, c_1, c_2), \pi_{\mathsf{vote}})$ and forms the array M. Note, the simulator uses the adversary-provided witness only if it does not have it already in its $\mathsf{Dec}_{\mathsf{int}}$ list (see line 40). It updates its $\mathsf{Dec}_{\mathsf{int}}$ and $\mathsf{VerEnc}_{\mathsf{int}}$ lists appropriately. If all m for $c \in B$ are non-$\perp$, it continues to follow the protocol code to compute $(\mathsf{CT}_1, \mathsf{CT}_2)$ and π_{agg} (it calls the adversary with $\mathcal{F}_{\mathsf{iNIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{F}_{\mathcal{R}_{\mathsf{AGG}}}].\mathsf{P.Prove}((\mathsf{pk}, \mathsf{CT}_1, \mathsf{CT}_2), B)$ to get π_{agg}). Note, the simulator additionally simulates $\mathcal{F}_{\mathcal{R}_{\mathsf{AGG}}}$ honestly in its simulation of $\mathcal{F}_{\mathsf{iNIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{F}_{\mathcal{R}_{\mathsf{AGG}}}].\mathsf{P.Prove}$. It updates its $\mathsf{Res}_{\mathsf{int}}$ list appropriately and returns $((\mathsf{CT}_1, \mathsf{CT}_2, \pi_{\mathsf{agg}}), M)$. Note if all c are well-formed (proof verifies), decryption will return a non-$\perp$ response that will match the response calculated using F in line 45 of $\mathcal{F}_{\mathsf{DHE}}$. The simulator also returns a pre-stored result if it exists. Thus the sanitization code of line 46 does not have any impact, and the code of $\mathcal{F}_{\mathsf{DHE}}$ and π_{DHE} for the Eval interface are identical.

$\mathcal{F}_{\mathsf{DHE}}$.P.Decrypt:

$\mathsf{Sim}(\mathsf{P.Decrypt}, \mathsf{CT})$: The simulator, between rounds T_2 and T_3, follows the code of π_{DHE} to compute CT' and π', and simulates $\mathcal{F}_{\mathsf{BC}}[\mathsf{P}, \mathbf{DC}, \Delta/2]$, $\mathcal{F}_{\mathsf{iNIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{F}_{\mathcal{R}_{\mathsf{AGG}}}]$ and $\mathcal{F}_{\mathsf{NIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{DDH}}]$ functionalities. It also needs to output M'. First in round T_2, it verifies the correctness of $\mathsf{CT} = (\mathsf{CT}_1, \mathsf{CT}_2, \pi_{\mathsf{agg}})$ via simulation of $\mathcal{F}_{\mathsf{iNIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{F}_{\mathcal{R}_{\mathsf{AGG}}}].\mathsf{P.Verify}((\mathsf{pk}, \mathsf{CT}_1, \mathsf{CT}_2), \pi_{\mathsf{agg}})$, wherein it receives B from the adversary. Further, it calls $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}].\mathsf{P.Verify}((\mathsf{pk}, c_1, c_2), \pi_{\mathsf{vote}})$ for each $c = (c_1, c_2, \pi_{\mathsf{vote}}) \in B$ to get m, which it adds to M'. Thus M', which is the set of plaintexts corresponding to B, is computed. Then, it performs the partial decryption of CT to produce d_{P} (as the simulator knows $\mathsf{sk}[\mathsf{P}]$) and further the proof π_{P} using $\mathcal{F}_{\mathsf{NIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{DDH}}].\mathsf{P.Prove}$. It broadcasts $(\mathsf{CT}, d_{\mathsf{P}}, \pi_{\mathsf{P}})$. Between rounds $T_2 + 1$ and $T_2 + \Delta/2$, it essentially simulates the network activity. Between rounds $T_2 + \Delta/2$ and T_3, it receives from all other committee members ($j \in [n]$) the $(\mathsf{CT}^{(j)}, d_j, \pi_j)$ tuple, and checks if its own CT matches the majority of the $\mathsf{CT}^{(j)}$'s. In case it matches, it sets $\mathsf{CT}' = \mathsf{CT}$, else it returns $\perp$. It verifies all received decryption proofs (π_j) to collect the necessary number of partial decryption shares (d_j) to interpolate D and further compute tally and proof. It sets $\pi' = \mathsf{proof}$. It sets $\mathsf{Res}_{\mathsf{int}}[\mathsf{pp}^*, \mathsf{CT}'] = \mathsf{tally}$ and $\mathsf{Ver}_{\mathsf{int}}[\mathsf{pp}^*, \mathsf{CT}', \mathsf{tally}, \pi'] = 1$.

$(\mathsf{CT}', M', \pi') \leftarrow \mathsf{Sim}(\mathsf{P.Decrypt}, \mathsf{CT})$: The simulator returns (CT', M', π') that it computed earlier. It updates its $\mathsf{Res}_{\mathsf{int}}$ and $\mathsf{Ver}_{\mathsf{int}}$ lists. Also, it always returns responses from the $\mathsf{Res}_{\mathsf{int}}$ list for a repeat input. As the simulator does decryption at most once, keeps the $\mathsf{Res}_{\mathsf{int}}$ and $\mathsf{Ver}_{\mathsf{int}}$ lists consistent, and the computation of tally via $F(M)$ gives the same result as obtained via actual decryption of CT', the sanitization code of $\mathcal{F}_{\mathsf{DHE}}$ (lines 56-73) does not modify res or lists Res and Ver. Thus the indistinguishability of $\mathcal{F}_{\mathsf{DHE}}$.P.Decrypt and π_{DHE}.P.Decrypt follows.

$\mathcal{F}_{\mathsf{DHE}}$.P.VerifyDec:

$b \leftarrow \mathsf{Sim}(\mathsf{P.VerifyDec}, \mathsf{pp}, \mathsf{CT}, \mathsf{res}, \pi)$: The simulator essentially follows the π_{DHE} code for the VerifyDec interface to obtain bit b which it returns. It updates its $\mathsf{Ver}_{\mathsf{int}}$ and $\mathsf{Res}_{\mathsf{int}}$ lists. Note, in the process it simulates $\mathcal{F}_{\mathsf{iNIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{F}_{\mathcal{R}_{\mathsf{AGG}}}].\mathsf{P.Verify}$ and $\mathcal{F}_{\mathsf{NIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{DDH}}].\mathsf{P.Verify}$. The simulator returns the already stored bit from the $\mathsf{Ver}_{\mathsf{int}}$ list for a repeat query

and keeps the lists consistent, thus the code of $\mathcal{F}_{\mathsf{DHE}}$ in lines 80-91 does not result in any overwriting of the returned response. Thus the indistinguishability of $\mathcal{F}_{\mathsf{DHE}}$.P.VerifyDec and π_{DHE}.P.VerifyDec follows.

As mentioned earlier, the simulator is also queried on the Verify interface of the $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}]$ functionality. We explain below how the simulator responds to it.

$\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}]$.P.Verify:

$w \leftarrow \mathsf{Sim}(\mathsf{P.Verify}, (\mathsf{pk}, c_1, c_2), \pi_{\mathsf{vote}})$: The simulator calls the adversary on $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}]$.P.Verify$((\mathsf{pk}, c_1, c_2), \pi_{\mathsf{vote}})$ to receive the witness (m, r). This witness is overwritten by $\mathsf{Wit}_{\mathsf{int}}[(\mathsf{pk}, c_1, c_2), \pi_{\mathsf{vote}}]$, if it exists, and returned. Else for a new statement (pk, c_1, c_2), $\mathsf{Wit}_{\mathsf{int}}$ is updated. With this corrected witness the functionality $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}$ computes the verification bit as 1, which matches the bit in $\mathsf{Ver}_{\mathsf{int}}$.

This completes the description of the simulator. We denote this simulator by $\mathsf{Sim}_{\mathsf{IDEAL}}$. Note this simulator follows the $(\mathcal{A}, \pi_{\mathsf{DHE}})$ code, including faithful simulation of the hybrids in all interfaces except in the $c \leftarrow \mathsf{Sim}(\mathsf{P.Encrypt}, \mathsf{pp}^*, 1^{|m|})$ interface. Here, the simulator, not knowing the input message m, encrypts a new valid message msg instead. In Lemma 1, we show that $\mathsf{Sim}_{\mathsf{IDEAL}}$ along with the ideal functionality $\mathcal{F}_{\mathsf{DHE}}$ code is indistinguishable from $(\mathcal{A}, \pi_{\mathsf{DHE}})$ code. We will refer to $(\mathcal{A}, \pi_{\mathsf{DHE}})$ code as $\mathsf{Sim}_{\mathsf{REAL}}$ in the following and it will additionally maintain all lists as in $\mathsf{Sim}_{\mathsf{IDEAL}}$. □

Lemma 1. *The ideal functionality $\mathcal{F}_{\mathsf{DHE}}$ with simulator $\mathsf{Sim}_{\mathsf{IDEAL}}$ is indistinguishable from the simulator $\mathsf{Sim}_{\mathsf{REAL}}$ code due to the IND-CPA security of ElGamal encryption.*

Proof. We start with a description of $\mathsf{Sim}_{\mathsf{REAL}}$ and then, via a sequence of intermediate hybrids Sim_1, Sim_2, Sim_3, Sim_4, Sim_5 and Sim_6, arrive at $\mathsf{Sim}_7 = \mathsf{Sim}_{\mathsf{IDEAL}}$, showing at each step how each simulator is indistinguishable from the previous one.

$\mathsf{Sim}_{\mathsf{REAL}}$: This is essentially the code described in Prot. 17 with the addition of all internal lists. The simulator knows the secret keys of all the participants via its execution of functionality 16. It does a faithful execution of all hybrids. It encrypts the input m even for the case $|\mathbf{DC} \cap \mathbf{C}| \leq t_{\mathsf{DC}}$. It computes the tally and proof via the decryption of the evaluated ciphertext.

Sim_1: This is similar to $\mathsf{Sim}_{\mathsf{REAL}}$ except that it skips the $\mathsf{R}(x, w)$ check in $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}]$.P.Prove and $\mathcal{F}_{\mathsf{NIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{DDH}}]$.P.Prove. Since the adversary in $\mathcal{F}_{\mathsf{NIZK}}$ is never passed the witness w, and proofs are computed on true statements, this simulator is indistinguishable from $\mathsf{Sim}_{\mathsf{REAL}}$.

Sim_2: This is similar to Sim_1 except that on a previously seen evaluated ciphertext CT in the Decrypt interface, it returns the message m stored in the $\mathsf{Dec}_{\mathsf{int}}$ list instead of actually decrypting it. Since the value stored in the $\mathsf{Dec}_{\mathsf{int}}$ list is the same as the decryption of the ciphertext, this simulator is indistinguishable from Sim_1.

Sim_3: This is similar to Sim_2 except that now it does not use the secret keys to decrypt the ciphertext in the Decrypt interface. Rather it uses the extractors from $\mathcal{F}_{\mathsf{iNIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{F}_{\mathcal{R}_{\mathsf{AGG}}}]$ and $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}]$ to compute the individual messages underlying CT and then sums them up to compute the tally. Additionally it does not use the secret key $\mathbf{sk}[j]$ to compute d_j. Instead it computes D using CT_2 and tally (see code below), and uses Lagrange interpolation in the exponent to compute them. Since the two methods result in the same tally and proofs, Sim_3 is indistinguishable from Sim_2.

Sim_3 follows the following code instead of lines 47-48 of protocol π_{DHE}:

1: Get B using $\mathcal{A}$ in $\mathcal{F}_{\mathsf{iNIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{F}_{\mathcal{R}_{\mathsf{AGG}}}]$.P.Verify$((\mathsf{pk}, \mathsf{CT}_1, \mathsf{CT}_2), \pi_{\mathsf{agg}})$
2: **for** $c = (c_1, c_2, \pi_{\mathsf{vote}}) \in B$ **do**
3: Get m using $\mathcal{A}$ in $\mathcal{F}_{\mathsf{NIZK}}^{\mathsf{gv}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{VOTE}}]$.P.Verify$((\mathsf{pk}, c_1, c_2), \pi_{\mathsf{vote}})$
4: Add m to M
5: $\mathsf{tally} \leftarrow F(M)$
6: $\mathsf{D} \leftarrow (\mathsf{CT}_2 / g^{\mathsf{tally}})$
7: Retrieve y_j and d_j for $j \in \mathbf{C}'$
8: $d_{\mathsf{P}} \leftarrow \mathsf{ExpLagrange}(t_{\mathsf{DC}}, \mathbf{C}' \cup \{0\}, \{d_j : j \in \mathbf{C}'\} \cup \{\mathsf{D}\})(\mathsf{P})$
9: $\pi_{\mathsf{P}} \leftarrow \mathcal{F}_{\mathsf{NIZK}}[\mathbf{P} \cup \mathbf{DC}, \mathcal{R}_{\mathsf{DDH}}]$.P.Prove$((g, \mathsf{CT}_1, y_{\mathsf{P}}, d_{\mathsf{P}}), \perp)$

Sim_4: Sim_4 differs from Sim_3 in the Setup interface. In particular in its simulation of the $\mathcal{F}_{\mathsf{KGen}}$ interface for the case when $|\mathbf{DC} \cap \mathbf{C}| \leq t_{\mathsf{DC}}$, the simulator takes an input public key $\mathsf{pk}_{\in}$ and ensures that $\mathsf{pk} = \mathsf{pk}_{\in}$, and the public parameter $\mathsf{pp} = (\mathsf{pk}, [y_j : j \in [n]])$ satisfies ppCheck.

The code that Sim_4 uses differs from the $\mathcal{F}_{\mathsf{KGen}}$ (functionality 15) code in lines 26-29. Instead of lines 26 and 27, the following code is executed, and code from lines 28 and 29 is ignored for this case ($|\mathbf{DC} \cap \mathbf{C}| \leq t_{\mathsf{DC}}$).

1: **if** $\mathbf{pk}[0] = \perp$ **then** $\mathbf{pk}[0] \xleftarrow{\$} \mathsf{pk}_{\in}$
2: $h \leftarrow \mathsf{ExpLagrange}(t_{\mathsf{KG}}, \mathbf{C}' \cup \{0\}, \{[g^{\mathbf{sk}[\mathsf{Q}]} : \mathsf{Q} \in \mathbf{C}'], \mathbf{pk}[0]\})$
3: **for** $\mathsf{Q} \in \mathbf{KC} \setminus \mathbf{C}'$ **do**
4: $\mathbf{pk}[\mathsf{Q}] \leftarrow h(\mathsf{Q})$

Also note that now Sim_4 does not know $\mathbf{sk}[\mathsf{Q}]$ for $\mathsf{Q} \in \mathbf{DC} \setminus \mathbf{C}'$, and therefore cannot return $\mathsf{sk}[\mathsf{P}]$ for $\mathsf{P} \in \mathbf{DC} \setminus \mathbf{C}'$ in line 31 of functionality $\mathcal{F}_{\mathsf{KGen}}$, but this being a local functionality the environment cannot distinguish.

Since this is a perfect simulation of the public parameters output by Setup, Sim_4 is indistinguishable from Sim_3.

Sim_5: This is similar to Sim_4 except that it encrypts an arbitrary valid message msg instead of m for the case $|\mathbf{DC} \cap \mathbf{C}| \leq t_{\mathsf{DC}}$. The indistinguishability of Sim_5 from Sim_4 follows from the IND-CPA security of ElGamal encryption. This is so because if any algorithm can distinguish between simulators Sim_4 and Sim_5, it can be used to solve the IND-CPA security, by setting $\mathsf{pk}_{\in}$ to be the public key received from the IND-CPA challenger and setting $m_0 = m$ and $m_1 = \mathsf{msg}$.

Sim_6: This is similar to Sim_5 except that it reverses the embedding of an input public key as done in going from Sim_3 to Sim_4. Thus it now simulates $\mathcal{F}_{\mathsf{KGen}}$ perfectly and knows

all the secret keys. Sim_6 is indistinguishable from Sim_5, since this is a perfect simulation of the public parameters output by Setup.

Sim_7: This is similar to Sim_6 except that it reverses the changes made in going from $\mathsf{Sim}_{\mathsf{REAL}}$ to Sim_1. Thus it does not skip any $\mathsf{R}(x, w)$ checks as it now knows all the witnesses. Sim_7 is indistinguishable from Sim_6, since the proofs involve all true statements.

Note, this simulator is identical to the one described above as $\mathsf{Sim}_{\mathsf{IDEAL}}$. Thus, the above sequence of simulators are identical to the environment as long as the IND-CPA security of ElGamal encryption holds. This concludes the proof of the lemma. □

APPENDIX E
BULLETIN BOARDS

As mentioned in the introduction, the $\mathcal{F}_{\mathsf{BB}}$ functionality (Func. 18) idealizes a weak bulletin board with receipts and proofs of inclusion, which makes it easier to implement by a group of BBMs (or a single BBM). The Setup interface models public-parameter generation, with *liveness* (non-$\perp$ pp^*) ensured when the BBM is honest. The Submit interface models the submission of data items to the bulletin board. If the BBM is honest, a valid submit results in a non-$\perp$ receipt, and the submitted data item is recorded in a set BB_{H}. The VerifyRec interface allows verification of receipts. If the BBM is honest and the verification is with respect to pp^*, then the validation bit b picked by the simulator for new receipts is overwritten by 0. This models the unforgeability of receipts.

In FormBB, the simulator outputs the final BB and a commitment μ to it. Furthermore, if the BBM is honest, the output BB must validate with respect to BB_{H}, and μ must be non-$\perp$. Users can ask to see the bulletin board from the BBM via the GetBB interface. Note that with a dishonest-majority BBM, each user may get a different bulletin board. This functionality is weak in that sense but can be realized with very weak assumptions. In the $\mathcal{F}_{\mathsf{L2Gov}}$ functionality, we add mechanisms to enforce a unique final BB. If the BBM is honest, the simulator-provided BB is ignored and overwritten by BB^*. The Com interface returns a commitment μ to any input BB.

The GetIncProof interface allows users to obtain a proof of inclusion of their data (Q, c) in the bulletin board. If (Q, c) is not in BB^*, the simulator-provided proof is ignored and $\perp$ is returned. The VerifyIncProof interface allows verification of received inclusion proofs. The simulator-provided validation bit b is overwritten by 0 when the requested proof is for commitment value μ^* and the vote (Q, c) is not in BB^*.

$\mathcal{F}_{\mathsf{DHE}}$ on its own does not suffice to enable voting: although it enables tallying of a set of encrypted votes, it does not allow their *collection*. We thus formulate a natural bulletin-board functionality $\mathcal{F}_{\mathsf{BB}}$ (Functionality 18) that enables this. Like $\mathcal{F}_{\mathsf{DHE}}$, $\mathcal{F}_{\mathsf{BB}}$ comes with a Setup interface that models system-parameter generation for a set of bulletin-board managers (BBMs). $\mathcal{F}_{\mathsf{BB}}$ also offers interfaces for submission of votes Submit, in response to which a receipt is obtained and the vote recorded, verification of receipts VerifyRec, formation and retrieval of a bulletin board FormBB and GetBB, as well as interfaces to obtain a proof of inclusion of data with respect to a fingerprint (commitment version) of the bulletin board GetIncProof and its verification VerifyIncProof. Once again, the simulator only needs to output a bit under the verification interfaces.

Importantly, $\mathcal{F}_{\mathsf{BB}}$ ensures the liveness of the bulletin board as long as the BBM is honest. This has three aspects. First, votes submitted in time will be responded to with a valid receipt. Second, such votes will be included in the final bulletin board according to a bulletin-board validity predicate $\mathsf{Val}_{\mathsf{BB}}$. The specific $\mathsf{Val}_{\mathsf{BB}}$ in our realization guarantees that all in-time votes will be included. Third, the bulletin board and proofs of inclusion can be computed by any honest BBM.

We describe the π_{BB} protocol in detail in Prot. 19. In brief, π_{BB} is a simple protocol in which parties submit data to a *single* bulletin-board manager, who in response issues receipts in the form of a digital signature. The bulletin board is formed by collecting all submitted data and computing a Merkle root of it as a short commitment. The Clean relation ensures that one vote from each voter is included in BB. The bulletin board can be retrieved by any party using the GetBB interface. Proofs of inclusion are simple Merkle paths, and verification checks this path against the commitment to the bulletin board.

S.Setup: The single BBM generates the parameters at time T_1 (thus no network activity) by invoking $\mathcal{F}_{\mathsf{Sig}}$.S.Gen, which outputs the public key used to sign receipts.

P.Submit(c): Parties can submit their messages between rounds T_0 and T_1. An authenticated channel $\mathcal{F}_{\mathsf{AC}}$ is used to send the message and fetch responses. On successful submission, the party obtains a signature from the $\mathbf{BBM}$ on the party's message.

P.VerifyRec$((Q, c), \rho)$: $\mathcal{F}_{\mathsf{Sig}}$.P.Verify is invoked to verify the received signature by each party.

S.FormBB: Between rounds T_2 and T_4, the $\mathbf{BBM}$ responds to incoming messages by adding messages to the bulletin board and issuing receipts. At round T_4, it commits to the bulletin board by outputting a Merkle root of the tree containing all the received messages. After round T_4, the $\mathbf{BBM}$ responds to incoming requests for the bulletin board or inclusion proofs.

P.GetBB: After round T_4, parties can demand access to the bulletin board or a commitment to the bulletin board, by submitting requests to the $\mathbf{BBM}$.

P.Com(BB): This is a local interface used by party P to compute a commitment to the bulletin board.

P.GetIncProof$(\mu, (Q, c))$: After round T_4, parties can demand access to inclusion proofs by submitting requests to the $\mathbf{BBM}$.

P.VerifyIncProof$(\mu, (Q, c), \pi)$: Inclusion proofs can be

locally verified by each party by evaluating the MerkleVerify function.

A. Security of π_{BB}

Theorem 4. *Let* $\mathbf{P}, \mathbf{BBM}, t_{\mathsf{BB}}, \mathsf{Val}_{\mathsf{BB}}, \Delta, \Delta_1, \ldots, \Delta_4$ *be a given set of parameters, where* $\mathbf{BBM} := \{\mathsf{S}\}$*, with* S *the identity of a single BBM server,* $t_{\mathsf{BB}} := 0$*,* $\mathsf{Val}_{\mathsf{BB}}(\mathsf{BB}, \mathsf{BB}_{\mathsf{H}}) := (\mathsf{BB} = \mathsf{Clean}(\mathsf{BB}, \mathsf{BB}_{\mathsf{H}}))$ *for a given relation* Clean *that is efficiently computable using the function* Cleanup*. Let* π_{BB} *be the protocol in Fig. 19 for this set of parameters and with respect to hybrids* $\mathcal{G}_{\mathsf{Clock}}[\mathbf{P}'_1]$*,* $\mathcal{F}_{\mathsf{PKI}}[\mathbf{P}'_2]$*,* $\mathcal{F}_{\mathsf{AC}}[\mathbf{P}'_3 \cup \mathbf{BBM}, \Delta/2]$*, and* $\mathcal{F}_{\mathsf{Sig}}[\mathbf{P}'_4, \{\mathsf{S}\}]$ *where* $\mathbf{P} \cup \mathbf{BBM} \subseteq \mathbf{P}'_1 \cap \mathbf{P}'_2 \cap \mathbf{P}'_3 \cap \mathbf{P}'_4 \wedge \mathbf{BBM} \cap \mathbf{P} = \varnothing$*, as well as secure algorithms* MerkleRoot*,* MerkleProof*, and* MerkleVerify*. Suppose further that all delay parameters are positive and* $\Delta \geq \Delta_3 - \Delta_4$*. Then* π_{BB} *UC-realizes* $\mathcal{F}_{\mathsf{BB}}$ *for the same set of parameters.*

At a high level, the simulator Sim internally runs copies of the hybrids $\mathcal{F}_{\mathsf{AC}}$, $\mathcal{F}_{\mathsf{Sig}}$, and $\mathcal{F}_{\mathsf{PKI}}$ and uses $\mathcal{G}_{\mathsf{Clock}}$ as provided. Sim, when run under $\mathcal{F}_{\mathsf{BB}}$, will aim to match the behavior of π_{BB} under various queries from the environment, essentially mimicking different interfaces of π_{BB} using the simulated hybrids. In doing this, Sim will also use a Merkle vector commitment scheme to compute and check proofs of inclusion. To show indistinguishability, we will argue that one can omit various pieces of code for output sanitization and consistency. This will typically be argued by relying on the properties of the underlying hybrids: for instance, that simulated outputs of hybrids will typically be already sanitized, that messages sent via $\mathcal{F}_{\mathsf{AC}}$ cannot be tampered with and are delivered in at most $\Delta/2$ rounds, or that, due to their binding property, it is infeasible to compute verifying Merkle proofs of incorrect statements.

Under full byzantine corruption, corrupted parties act as forwarders to the adversary, taking its instructions to forward inputs to other machines. The simulator corrupts the same set of parties that the adversary does. Furthermore, since corruptions are static, the simulator need not simulate any state for corrupted parties (since they are empty to begin with). The simulator relays the communication between the adversary and the corrupted parties. (Due to PID-wise connectivity and PID-wise corruptions, essentially all machines with a corrupted PID can be seen as one and absorbed into the adversary.) The only interaction between a corrupted party and a non-corrupted machine is between a corrupted dummy party and its ideal functionality. (Without loss of generality, the environment does not call any corrupted parties or assume their role.) The simulator will also relay these to its internally simulated functionalities, or to the provided (global) functionalities, as appropriate.

Next, we describe the operation of the simulator Sim for our π_{BB}. To this end, we specify its behavior under each of its invocations under $\mathcal{F}_{\mathsf{BB}}$. We assume, wlog, that the adversary is a dummy adversary and acts as a forwarder for the environment.

1st Sim(P.Setup): The simulator returns ok in rounds before T_1 (there is no activity in π_{BB}).

2nd Sim(P.Setup): The simulator in round T_1 generates a verification key vk^* as the public parameters through the P.Gen interface of the internally simulated $\mathcal{F}_{\mathsf{Sig}}$. Sim asks the adversary for a verification key vk^*, and, if needed, sanitizes it as per the specification of $\mathcal{F}_{\mathsf{Sig}}$. Sim returns the final (possibly sanitized) vk^* as the public parameters. The simulator then registers this key with the internally run $\mathcal{F}_{\mathsf{PKI}}$ on behalf of the BBM. Note that when Sim is run within $\mathcal{F}_{\mathsf{BB}}$, because vk^* is already sanitized, the code for further sanitization of pp in the honest-BBM case has no further effect and can be dropped. Furthermore, since there is a single BBM, there is always agreement on vk^*. Thus, the codes of $\mathcal{F}_{\mathsf{BB}}$ and π_{BB} are identical for the Setup interface.

Sim(P.Submit, c): Sim internally simulates $\mathcal{F}_{\mathsf{AC}}$.P.Send on $(\mathsf{S}, (\text{“vote”}, c))$, informing the adversary as in the specification of $\mathcal{F}_{\mathsf{AC}}$ and appending outputs to an internal list. (Note that Sim can obtain the current round number by calling the T interface of $\mathcal{G}_{\mathsf{Clock}}$.) The simulator then runs $\mathcal{F}_{\mathsf{AC}}$.P.Fetch, simulating Δ-old message indices internally and asking the adversary for indices of early messages. It delivers messages from the server as specified by these indices. Looking at the simulation under FormBB, these will contain signatures (as receipts), implemented via a simulated signature functionality. The simulator verifies the receipts using vk^* from the setup phase. We will argue below, under the simulation of VerifyRec, that entries on lists VerRec and Ver (of $\mathcal{F}_{\mathsf{Sig}}$) are synchronized and that bookkeeping using VerRec can be removed. Note that $\mathcal{F}_{\mathsf{BB}}$ contains a delay parameter of Δ, meaning that if the BBM is honest, only votes that are submitted before $T_3 - \Delta$ are guaranteed to be included and a receipt received. This is achieved by π_{BB}: as long as a vote is submitted Δ away from T_3, since the network is $\Delta/2$-synchronous, the BBM will receive it in time. Furthermore, a receipt for it will be generated and delivered within another $\Delta/2$ rounds. Thus, the codes of $\mathcal{F}_{\mathsf{BB}}$ and π_{BB} are identical for the Submit interface.

Sim(P.VerifyRec, $(\mathsf{Q}, c), \rho$): The simulator internally runs the Verify interface of $\mathcal{F}_{\mathsf{Sig}}$ using vk^* from the setup phase. Note that verification outputs are already sanitized in this simulation. Moreover, entries on the Ver list (as part of the $\mathcal{F}_{\mathsf{Sig}}$ simulation) and VerRec list (as part of $\mathcal{F}_{\mathsf{BB}}$) are ‘in sync’ in the sense that there is a signature entry on one iff there is a receipt entry on the other. This is because receipt generation and verification affect VerRec, and signature generation and verification affect Ver. Note also that the Verify interface of $\mathcal{F}_{\mathsf{Sig}}$ sets the verification bit to 0 for new entries if the signer is honest, which is what $\mathcal{F}_{\mathsf{BB}}$ does when the single BBM is honest.

Functionality 18: $\mathcal{F}_{\mathsf{BB}}[\mathbf{P}, \mathbf{BBM}, t_{\mathsf{BB}}, \mathsf{Val}_{\mathsf{BB}}, \Delta, \Delta_1, \ldots, \Delta_4]$

Parameters

Hybrids: $\mathcal{G}_{\mathsf{Clock}}[\mathbf{P}']$
Requires: $\mathbf{P} \cup \mathbf{BBM} \subseteq \mathbf{P}' \wedge \mathbf{BBM} \cap \mathbf{P} = \varnothing$

Initialize()

$\mathsf{BB}^*, \mu^*, \mathsf{pp}^* \leftarrow \bot$
$\mathsf{done}_1, \mathsf{done}_2, \mathsf{SubTime}[*,*] \leftarrow \bot$
$\mathsf{VerRec}[*,*], \mathsf{VerInc}[*,*,*], \mu[*], \mathsf{Set}[*] \leftarrow \bot$
$R_1, R_2, \mathsf{BB}_{\mathsf{H}} \leftarrow \varnothing$
$T_0 \leftarrow \mathsf{T}()$
for $i \in [1,2,3]$ **do** $T_i \leftarrow T_{i-1} + \Delta_i$
$T_4 \leftarrow T_2 + \Delta_4$

P.Setup() for $\mathsf{P} \in \mathbf{BBM}$

if $\mathsf{T}() < T_0$ **then return** $\bot$
if $T_0 \le \mathsf{T}() < T_1$ **then**
  Sim(P.Setup)
  return $\bot$
if $\mathsf{T}() = T_1$ **then**
  $\mathsf{pp}' \leftarrow$ Sim(P.Setup)
  if $\neg\mathsf{done}_1$ **then** $(\mathsf{pp}^*, \mathsf{done}_1) \leftarrow (\mathsf{pp}', \top)$
  if $|\mathbf{BBM} \cap \mathbf{C}| \le t_{\mathsf{BB}}$ **then**
    $\mathsf{San}(\mathsf{pp}^*)$: $\mathsf{pp}^* \neq \bot$
  return pp^*
if $T_1 < \mathsf{T}()$ **then return** $\bot$

P.Submit(c) for $\mathsf{P} \in \mathbf{P}$

if $\mathsf{T}() < T_2$ **then return** $\bot$
if $T_2 \le \mathsf{T}() < T_3$ **then**
  $\rho \leftarrow$ Sim(P.Submit, c)
  if $|\mathbf{BBM} \cap \mathbf{C}| \le t_{\mathsf{BB}} \wedge \mathsf{T}() < T_3 - \Delta$ **then**
    if $(\mathsf{P}, c) \notin \mathsf{BB}_{\mathsf{H}}$ **then**
      $\mathsf{BB}_{\mathsf{H}} \leftarrow \mathsf{BB}_{\mathsf{H}} \cup \{(\mathsf{P}, c)\}$
      $\mathsf{SubTime}[\mathsf{P}, c] \leftarrow \mathsf{T}()$
  if $|\mathbf{BBM} \cap \mathbf{C}| \le t_{\mathsf{BB}} \wedge \mathsf{T}() - \mathsf{SubTime}[\mathsf{P}, c] \ge \Delta$ **then**
    $\mathsf{San}(\rho)$: $\mathsf{VerRec}[(\mathsf{P}, c), \rho] \neq 0 \wedge \rho \neq \bot$
    $\mathsf{VerRec}[(\mathsf{P}, c), \rho] \leftarrow 1$
    return ρ
  if $\mathsf{VerRec}[(\mathsf{P}, c), \rho] = 0$ **then return** $\bot$
  $\mathsf{VerRec}[(\mathsf{P}, c), \rho] \leftarrow 1$
  return ρ
if $T_3 \le \mathsf{T}()$ **then return** $\bot$

P.VerifyRec($(\mathsf{Q}, c), \rho$) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{BBM}$; if gv: any sid

$b \leftarrow$ Sim(P.VerifyRec, $(\mathsf{Q}, c), \rho$)
$\mathsf{San}(b)$: $b \in \{0, 1\}$
if $\mathsf{VerRec}[(\mathsf{Q}, c), \rho] \neq \bot$ **then return** $\mathsf{VerRec}[(\mathsf{Q}, c), \rho]$
if $|\mathbf{BBM} \cap \mathbf{C}| \le t_{\mathsf{BB}}$ **then** $b \leftarrow 0$
$\mathsf{VerRec}[(\mathsf{Q}, c), \rho] \leftarrow b$
return b

P.FormBB() for $\mathsf{P} \in \mathbf{BBM}$, once

if $\mathsf{T}() < T_2$ **then return** $\bot$
if $T_2 \le \mathsf{T}() < T_4$ **then**
  Sim(P.FormBB)
  return $\bot$
if $\mathsf{T}() = T_4$ **then**
  $(\mathsf{BB}', \mu') \leftarrow$ Sim(P.FormBB)
  if $\neg\mathsf{done}_2$ **then** $(\mathsf{BB}^*, \mu^*, \mathsf{done}_2) \leftarrow (\mathsf{BB}', \mu', \top)$
  if $|\mathbf{BBM} \cap \mathbf{C}| \le t_{\mathsf{BB}}$ **then**
    $\mathsf{San}(\mathsf{BB}^*)$: $\mathsf{Val}_{\mathsf{BB}}(\mathsf{BB}^*, \mathsf{BB}_{\mathsf{H}})$
  if $\mathsf{BB}^* = \bot$ **then return** $(\bot, \bot)$
  if $\mu[\mathsf{BB}^*] \neq \bot$ **then return** $(\mathsf{BB}^*, \mu[\mathsf{BB}^*])$
  $\mathsf{San}(\mu^*)$: $\mu^* \neq \bot \wedge (\forall \mathsf{BB}' \neq \mathsf{BB}^*)\mu[\mathsf{BB}'] \neq \mu^*$
  $\mu[\mathsf{BB}^*] \leftarrow \mu^*$; $\mathsf{Set}[\mu^*] \leftarrow \mathsf{BB}^*$
  return (BB^*, μ^*)
if $T_4 < \mathsf{T}()$ **then**
  Sim(P.FormBB)
  return $\bot$

P.GetBB() for $\mathsf{P} \in \mathbf{P}$

if $\mathsf{T}() \le T_4$ **then return** $\bot$
if $T_4 < \mathsf{T}()$ **then**
  $\mathsf{BB} \leftarrow$ Sim(P.GetBB)
  if $|\mathbf{BBM} \cap \mathbf{C}| \le t_{\mathsf{BB}}$ **then**
    $R_1 \leftarrow R_1 \cup \{(\mathsf{P}, \mathsf{T}())\}$
    $t_0 \leftarrow \min\{t : (\mathsf{P}, t) \in R_1\}$
    if $\mathsf{T}() - t_0 \ge \Delta$ **then**
      $\mathsf{BB} \leftarrow \mathsf{BB}^*$
      $R_1 \leftarrow R_1 \setminus \{(\mathsf{P}, t) \in R_1 : t \ge t_0\}$
  return BB

P.Com(BB) for $\mathsf{P} \in \mathbf{P}$; if gv: any sid

$\mu \leftarrow$ Sim(P.Com, BB)
if $\mu[\mathsf{BB}] \neq \bot$ **then**
  return $\mu[\mathsf{BB}]$
$\mathsf{San}(\mu)$: $\mu \neq \bot \wedge (\forall \mathsf{BB}' \neq \mathsf{BB})\mu[\mathsf{BB}'] \neq \mu$
$\mathsf{Set}[\mu] \leftarrow \mathsf{BB}$;$\mu[\mathsf{BB}] \leftarrow \mu$
return $\mu[\mathsf{BB}]$

P.GetIncProof($\mu, (\mathsf{Q}, c)$) for $\mathsf{P} \in \mathbf{P}$

if $\mathsf{T}() \le T_4$ **then return** $\bot$
if $T_4 < \mathsf{T}()$ **then**
  $\pi \leftarrow$ Sim(P.GetIncProof, $\mu, (\mathsf{Q}, c)$)
  if $\mathsf{Set}[\mu] = \bot \vee (\mathsf{Q}, c) \notin \mathsf{Set}[\mu]$ **then**
    return $\bot$
  if $|\mathbf{BBM} \cap \mathbf{C}| \le t_{\mathsf{BB}}$ **then**
    $R_2 \leftarrow R_2 \cup \{(\mathsf{P}, \mathsf{Q}, c, \mathsf{T}())\}$
    $t_0 \leftarrow \min\{t : (\mathsf{P}, \mathsf{Q}, c, t) \in R_2\}$
    if $\mathsf{T}() - t_0 \ge \Delta$ **then**
      $\mathsf{San}(\pi)$: $\pi \neq \bot \wedge \mathsf{VerInc}[\mu, (\mathsf{Q}, c), \pi] \neq 0$
      $\mathsf{VerInc}[\mu, (\mathsf{Q}, c), \pi] \leftarrow 1$
      $R_2 \leftarrow R_2 \setminus \{(\mathsf{P}, \mathsf{Q}, c, t) \in R_2 : t \ge t_0\}$
      return π
  if $\mathsf{VerInc}[\mu, (\mathsf{Q}, c), \pi] = 0$ **then return** $\bot$
  $\mathsf{VerInc}[\mu, (\mathsf{Q}, c), \pi] \leftarrow 1$
  return π

P.VerifyIncProof($\mu, (\mathsf{Q}, c), \pi$) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{BBM}$; if gv: any sid

$b \leftarrow$ Sim(P.VerifyIncProof, $\mu, (\mathsf{Q}, c), \pi$)
$\mathsf{San}(b)$: $b \in \{0, 1\}$
if $\mathsf{VerInc}[\mu, (\mathsf{Q}, c), \pi] \neq \bot$ **then**
  return $\mathsf{VerInc}[\mu, (\mathsf{Q}, c), \pi]$
if $\mathsf{Set}[\mu] \neq \bot \wedge (\mathsf{Q}, c) \notin \mathsf{Set}[\mu]$ **then**
  $b \leftarrow 0$
$\mathsf{VerInc}[\mu, (\mathsf{Q}, c), \pi] \leftarrow b$
return $\mathsf{VerInc}[\mu, (\mathsf{Q}, c), \pi]$

Accordingly, the checks after running the simulator under VerifyRec can be dropped, and the codes of this interface under $\mathcal{F}_{\mathsf{BB}}$ and π_{BB} are identical.

1st Sim(P.FormBB): The simulator simulates the collection of submissions and receipt generation. It internally simulates Fetch, Sign, Send to collect submitted c's, sign them using $\mathcal{F}_{\mathsf{Sig}}$, and send receipts back to the submitter via $\mathcal{F}_{\mathsf{AC}}$. It collects (P, c) in a set BB^*. Note that when the BBM is honest, BB^* will contain all (P, c) that were sent at least Δ rounds before T_3. This is because $\mathcal{F}_{\mathsf{AC}}$ is authenticated and at most $\Delta/2$-delayed.

2nd Sim(P.FormBB): The simulator in round T_4 finalizes

Protocol 19: $\pi_{\mathsf{BB}}[\mathbf{P}, \mathbf{BBM} := \{\mathsf{S}\}, t_{\mathsf{BB}} := 0, \mathsf{Val}_{\mathsf{BB}}, \Delta, \Delta_1, \ldots, \Delta_4]$

Parameters

Hybrids: $\mathcal{G}_{\mathsf{Clock}}[\mathbf{P}'_1]$, $\mathcal{F}_{\mathsf{PKI}}[\mathbf{P}'_2]$, $\mathcal{F}_{\mathsf{AC}}[\mathbf{P}'_3, \Delta/2]$, $\mathcal{F}_{\mathsf{Sig}}[\mathbf{P}'_4, \{\mathsf{S}\}]$
Requires: $\mathbf{P} \cup \mathbf{BBM} \subseteq \mathbf{P}'_1 \cap \mathbf{P}'_2 \cap \mathbf{P}'_3 \cap \mathbf{P}'_4 \wedge \mathbf{BBM} \cap \mathbf{P} = \varnothing$

Proc. P.Initialize() for $\mathsf{P} \in \mathbf{P} \cup \mathbf{BBM}$

Register with $\mathcal{G}_{\mathsf{Clock}}$
if $\mathsf{P} = \mathsf{S}$ **then** $\mathsf{BB}^*, \mu^*, \mathsf{vk}^* \leftarrow \bot$
$T_0 \leftarrow \mathsf{T}()$
for $i \in [1, 2, 3]$ **do** $T_i \leftarrow T_{i-1} + \Delta_i$
$T_4 \leftarrow T_2 + \Delta_4$

S.Setup() for $\mathsf{S} \in \mathbf{BBM}$

if $\mathsf{T}() < T_0$ **then return** $\bot$
if $T_0 \leq \mathsf{T}() < T_1$ **then**
  if $\mathsf{vk}^* = \bot$ **then** $\mathsf{vk}^* \leftarrow \mathcal{F}_{\mathsf{Sig}}.\mathsf{S.Gen}()$
  $\mathsf{pp}^* \leftarrow \mathcal{F}_{\mathsf{PKI}}.\mathsf{P.Setup}()$
  $\mathsf{cert} \leftarrow \mathcal{F}_{\mathsf{PKI}}.\mathsf{P.Register}(\mathsf{pp}^*, \mathsf{vk}^*)$
  return $\bot$
if $\mathsf{T}() = T_1$ **then return** vk^*
if $T_1 < \mathsf{T}()$ **then return** $\bot$

P.Submit(c) for $\mathsf{P} \in \mathbf{P}$

if $\mathsf{T}() < T_2$ **then return** $\bot$
if $T_2 \leq \mathsf{T}() < T_3$ **then**
  $\mathcal{F}_{\mathsf{AC}}.\mathsf{P.Fetch}()$
  $\mathcal{F}_{\mathsf{AC}}.\mathsf{P.Send}(\mathsf{S}, (\text{“vote”}, c))$
  $\mathsf{msgs} \leftarrow \mathsf{msgs} : \mathcal{F}_{\mathsf{AC}}.\mathsf{P.Fetch}()$
  $L_1 := [(t, \sigma)] \leftarrow \mathsf{msgs}|\mathrm{grep}(\mathsf{S}, \mathsf{P}, *, (\text{“sig”}, (\mathsf{P}, c), *))$
  $\mathsf{vk}^* \leftarrow \mathcal{F}_{\mathsf{PKI}}.\mathsf{P.Retrieve}(\mathsf{pp}^*, \mathsf{S})$
  if $\neg \mathcal{F}_{\mathsf{Sig}}.\mathsf{P.Verify}(\mathsf{vk}^*, (\mathsf{P}, c), \sigma)$ **then return** $\bot$
  return $\mathsf{head}(L_1.\sigma)$
if $T_3 \leq \mathsf{T}()$ **then return** $\bot$

P.VerifyRec($(\mathsf{Q}, c), \rho$) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{BBM}$; if gv: any sid

$\mathsf{vk}^* \leftarrow \mathcal{F}_{\mathsf{PKI}}.\mathsf{P.Retrieve}(\mathsf{pp}^*, \mathsf{S})$
return $\mathcal{F}_{\mathsf{Sig}}.\mathsf{P.Verify}(\mathsf{vk}^*, (\mathsf{Q}, c), \rho)$

S.FormBB() for $\mathsf{S} \in \mathbf{BBM}$

if $\mathsf{T}() < T_2$ **then return** $\bot$
if $T_2 \leq \mathsf{T}() < T_4$ **then**
  $\mathsf{msgs} \leftarrow \mathsf{msgs} : \mathcal{F}_{\mathsf{AC}}.\mathsf{S.Fetch}()$
  $L_1 := [(\mathsf{P}, t, c)] \leftarrow \mathsf{msgs}|\mathrm{grep}(*, \mathsf{S}, *, (\text{“vote”}, *))$
  for $(\mathsf{P}, t, c) \in L_1$ **do**
    $\mathsf{BB}^* \leftarrow \mathsf{BB}^* \cup \{(\mathsf{P}, c)\}$
    $\sigma \leftarrow \mathcal{F}_{\mathsf{Sig}}.\mathsf{S.Sign}((\mathsf{P}, c))$
    $\mathcal{F}_{\mathsf{AC}}.\mathsf{S.Send}(\mathsf{P}, (\text{“sig”}, (\mathsf{P}, c), \sigma))$
  return $\bot$
if $\mathsf{T}() = T_4$ **then**
  $\mathsf{BB}^* \leftarrow \mathsf{Cleanup}(\mathsf{BB}^*)$
  $\mu^* \leftarrow \mathsf{MerkleRoot}(\mathsf{BB}^*)$
  return (BB^*, μ^*)
if $T_4 < \mathsf{T}()$ **then**
  $\mathsf{msgs} \leftarrow \mathsf{msgs} : \mathcal{F}_{\mathsf{AC}}.\mathsf{S.Fetch}()$
  $L_1 := [(\mathsf{P}, t)] \leftarrow \mathsf{msgs}|\mathrm{grep}(*, \mathsf{S}, \text{“bb-req”})$
  $L_3 := [(\mathsf{P}, t, (\mathsf{Q}, c))] \leftarrow \mathsf{msgs}|\mathrm{grep}(*, \mathsf{S}, \text{“proof-req”})$
  for $\mathsf{P} \in L_1.P$ **do**
    $\mathcal{F}_{\mathsf{AC}}.\mathsf{S.Send}(\mathsf{P}, (\text{“bb”}, \mathsf{BB}^*))$
  for $(\mathsf{P}, (\mathsf{Q}, c)) \in L_3.(\mathsf{P}, (\mathsf{Q}, c))$ **do**
    $\pi \leftarrow \mathsf{MerkleProof}(\mathsf{BB}^*, (\mathsf{Q}, c))$
    $\mathcal{F}_{\mathsf{AC}}.\mathsf{S.Send}(\mathsf{P}, (\text{“proof”}, \mathsf{Q}, c, \pi))$
  $\mathcal{F}_{\mathsf{AC}}.\mathsf{S.Fetch}()$
  return $\bot$

P.GetBB() for $\mathsf{P} \in \mathbf{P}$

if $\mathsf{T}() < T_4$ **then return** $\bot$
if $T_4 \leq \mathsf{T}()$ **then**
  $\mathcal{F}_{\mathsf{AC}}.\mathsf{P.Send}(\mathsf{S}, \text{“bb-req”})$
  $\mathsf{msgs} \leftarrow \mathsf{msgs} : \mathcal{F}_{\mathsf{AC}}.\mathsf{P.Fetch}()$
  $L_1 := [(t, \mathsf{BB})] \leftarrow \mathsf{msgs}|\mathrm{grep}(\mathsf{S}, \mathsf{P}, *, (\text{“bb”}, *))$
  return $\mathsf{head}(L_1.\mathsf{BB})$

P.Com(BB) for $\mathsf{P} \in \mathbf{P}$; if gv: any sid

return $\mathsf{MerkleRoot}(\mathsf{BB})$

P.GetIncProof(μ, Q, c) for $\mathsf{P} \in \mathbf{P}$

if $\mathsf{T}() \leq T_4$ **then return** $\bot$
if $T_4 < \mathsf{T}()$ **then**
  $\mathcal{F}_{\mathsf{AC}}.\mathsf{P.Send}(\mathsf{S}, (\text{“proof-req”}, \mu, (\mathsf{Q}, c)))$
  $\mathsf{msgs} \leftarrow \mathsf{msgs} : \mathcal{F}_{\mathsf{AC}}.\mathsf{P.Fetch}()$
  $L_1 := [(t, \pi)] \leftarrow \mathsf{msgs}|\mathrm{grep}(\mathsf{S}, \mathsf{P}, *, (\text{“proof”}, \mathsf{Q}, c, *))$
  $\mathsf{MerkleVerify}(\mu, (\mathsf{Q}, c), \mathsf{head}(L_1.\pi))$
  return $\mathsf{head}(L_1.\pi)$

P.VerifyIncProof($\mu, (\mathsf{Q}, c), \pi$) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{BBM}$; if gv: any sid

return $\mathsf{MerkleVerify}(\mu, (\mathsf{Q}, c), \pi)$

BB^* as the bulletin board and computes its root μ^* via a Merkle root computation with respect to BB^*. Note also that there is agreement on BB^* since there is a single BBM. Due to the binding property, μ^* will be distinct from the roots of other BB's. Finally, BB^* is computed via the Cleanup operation. Due to the authenticity and synchronicity properties of the network, all submitted data that are submitted Δ away from T_3 are passed to Cleanup. Note that the $\mathsf{Val}_{\mathsf{BB}}$ relation in $\mathcal{F}_{\mathsf{BB}}$ is defined using Cleanup. Thus, there is no need to sanitize BB^* due to an invalid $\mathsf{Val}_{\mathsf{BB}}$.

3rd Sim(P.FormBB): The simulator simulates sending the BB and proofs of inclusion. It internally simulates Fetch and Send to deal with BB^*, μ^*, and π requests, where π is computed via Merkle proofs as in the specification of π_{BB}. Since $\mathcal{F}_{\mathsf{AC}}$ is authenticated, correct values are sent back in at most $\Delta/2$ rounds. This ensures parties receive responses within Δ as specified in $\mathcal{F}_{\mathsf{BB}}$.

Sim(P.GetBB): The simulator runs $\mathcal{F}_{\mathsf{AC}}$.P.Send with the BBM as the destination using "bb-req" as the message. The messages will be delivered to the BBM in at most $\Delta/2$ rounds. The simulator then runs Fetch, asking the adversary for delivery of messages. It then returns BB, which will be delivered in at most Δ rounds after they are received. Thus, these requests are answered in at most Δ rounds.

Sim(P.Com, BB): The simulator computes the Merkle root μ of the input BB and returns it. Due to the binding property of Merkle roots, no two distinct BB's will have

the same μ (except with small probability). Thus, the output μ does not need to be sanitized.

$\mathsf{Sim}(\mathsf{P.GetIncProof}, \mu, (\mathsf{Q}, c))$: The simulator runs $\mathcal{F}_{\mathsf{AC}}$.P.Send with "proof-req" and the provided (Q, c) to the BBM as the destination. It then (internally) fetches from the server for $(\text{"proof"}, (\mathsf{Q}, c, *))$ and returns the proof. This proof is computed via the Merkle proof path algorithm with respect to the unique set associated with μ (which is kept track of internally). Due to the correctness and soundness of Merkle, this proof will always verify, and does not need to be sanitized. Once again, when the BBM is honest, proofs will be delivered within at most Δ rounds after requesting them, since the network is authenticated and $\Delta/2$-synchronous.

$\mathsf{Sim}(\mathsf{P.VerifyIncProof}, \mu, (\mathsf{Q}, c), \pi)$: The simulator verifies $(\mu, (\mathsf{Q}, c), \pi)$ using the Merkle proof verification algorithm to respond with a decision bit b. Due to the determinism of Merkle proofs and the fact that correctly generated proofs always verify, the answers provided by the simulator will be consistent with previous answers. Thus, checks in $\mathcal{F}_{\mathsf{BB}}$ for consistency with the previous answers can be dropped. Moreover, since Merkle proofs are sound, there will not arise a situation where a proof for an incorrect statement is accepted (except with small probability). Thus, checks in VerifyIncProof of $\mathcal{F}_{\mathsf{BB}}$, where a bit 1 output by the simulator is overwritten by 0 when $(\mathsf{Q}, c) \notin \mathsf{BB}^*$, can be removed. This means that the output of the simulator will be used as is in $\mathcal{F}_{\mathsf{BB}}$. Hence, this interface of $\mathcal{F}_{\mathsf{BB}}$ when run with the simulator and that of π_{BB} are functionally identical.

$\mathsf{Sim}(\mathsf{P.OK})$ from $\mathcal{G}_{\mathsf{Clock}}$: The simulator forwards the call to the adversary and relays back the response (an ok) to $\mathcal{G}_{\mathsf{Clock}}$.

Overall, execution of $\mathcal{F}_{\mathsf{BB}}$ with the simulator Sim above is indistinguishable, from any interactive distinguisher's $\mathcal{E}$ point of view, from that of π_{BB}.

Appendix F
Layer-2 Governance

We now present the pseudo-code of the $\mathcal{F}_{\mathsf{L2Gov}}$ functionality and π_{L2Gov} protocol. A compact description of both was presented in Sect. VI.

As mentioned, the $\mathcal{F}_{\mathsf{L2Gov}}$ functionality shares several interfaces with $\mathcal{F}_{\mathsf{DHE}}$ and $\mathcal{F}_{\mathsf{BB}}$. $\mathcal{F}_{\mathsf{L2Gov}}$ comes with two set-ups, BBSetup and VCSetup, which are essentially the same as the $\mathcal{F}_{\mathsf{BB}}$.Setup and $\mathcal{F}_{\mathsf{DHE}}$.Setup interfaces, respectively. The Vote interface allows any voter in $\mathbf{P}$ to submit an encrypted ballot to layer-1 or layer-2. Thus, the pseudo-code is shared with that of the $\mathcal{F}_{\mathsf{DHE}}$.Encrypt and $\mathcal{F}_{\mathsf{BB}}$.Submit interfaces. As layer-1 is assumed to be reliable and censorship-resistant, all ballots submitted to layer-1 are added to a list V_1, while ballots submitted to layer-2 are added to V_2 only if the $\mathbf{BBM}$ is t_{BB}-honest.

The FinalizeVotes interface is similar to the FormBB interface of $\mathcal{F}_{\mathsf{BB}}$ (it models both finalizing the list of votes and servicing incoming requests for it), but here it computes a finalized list V^* of collected votes from both layer-1 and layer-2. Liveness guarantees are captured via the predicates $\mathsf{Val}^1_{\mathsf{BB}}$ and $\mathsf{Val}^2_{\mathsf{BB}}$.

The CheckInc interface combines the GetIncProof and VerifyIncProof interfaces of $\mathcal{F}_{\mathsf{BB}}$ into one. Thus, on receiving the validation bit b from the simulator, the functionality decides whether b needs to be overwritten with 0 if the decrypted item associated with the submitted ciphertext is not in V^*. The Complain interface allows any party to file a complaint, potentially on behalf of others, by submitting a ballot. This can be invoked if a vote-inclusion check via CheckInc fails. The functionality keeps track of pending complaints in a list Com. The Resolve interface allows a BBM to resolve a pending complaint: the vote underlying the ballot is recovered, and the complaint is resolved and removed from Com only if it appears in V^*.

The Tally interface idealizes the VC's ability to form the tally tally^* over the finalized list of votes. The tally is computed via a tally function F, which is a parameter of the system and operates over V^*. If the simulator-provided tally' differs from tally^*, $\perp$ is returned.

We now describe our protocol π_{L2Gov} (see Prots. 21 and 23 for the pseudo-code). This protocol uses distributed homomorphic encryption $\mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}$, bulletin board $\mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}$, the mainchain and clock functionalities $\mathcal{G}_{\mathsf{MC}}$ and $\mathcal{G}_{\mathsf{Clock}}$, a PKI functionality $\mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}$, as well as $\mathcal{F}^{\mathsf{gv}}_{\mathsf{Sig}}$ and $\mathcal{F}^{\mathsf{gv}}_{\mathsf{NIZK}}$ as hybrids. In our realization, $\mathbf{VC}$ is a small- to medium-sized committee, while there is a single $\mathbf{BBM}$. Our protocol comprises an off-chain component (Prot. 21), in which the $\mathbf{VC}$, $\mathbf{BBM}$, and voters interact without using the mainchain. It also has an on-chain component GovSC (Prot. 23), implemented via a contract deployed on the mainchain. Relation $\mathcal{R}_{\mathsf{BBM}}$, used by the $\mathbf{BBM}$, is described in Relation 22.

$\mathbf{VC}$ and $\mathbf{BBM}$ use the respective Setup interfaces of $\mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}$ and $\mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}$ to generate their public parameters. They also each generate a signing key and register it with a PKI. Specifically, functionality $\mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}$ is used to bind the short-term signature verification keys of various parties to their identities (e.g., long-term on-chain keys). The use of $\mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}$ abstracts away how this binding is achieved (e.g., through $\mathcal{G}_{\mathsf{MC}}$ itself). The $\mathbf{BBM}$ then deploys a GovSC contract on the mainchain, along with various initial data, such as Merkle roots of the identities of the $\mathbf{VC}$, $\mathbf{BBM}$, and voters. We assume that this data is common knowledge and is agreed upon through an out-of-band mechanism (e.g., using the blockchain itself). Thus, different parties can verify that they use the correct contract, and we also parameterize our functionality with respect to them.

The $\mathbf{BBM}$ signs and submits its public parameters to a GovSC contract, as does the $\mathbf{VC}$. The contract accepts these if there are sufficiently many signatures on them, according to two threshold values t_{BB} and t_{VC}.

Functionality 20: $\mathcal{F}_{\mathsf{L2Gov}}[\mathbf{P}, \mathbf{VC}, \mathbf{BBM}, t_{\mathsf{VC}}, t_{\mathsf{BB}}, \mathsf{Val}, \mathsf{Val}^1_{\mathsf{BB}}, \mathsf{Val}^2_{\mathsf{BB}}, F, \ \Delta, \Delta_1, \ldots, \Delta_9]$

Parameters

Hybrids: $\mathcal{G}_{\mathsf{Clock}}[\mathbf{P}']$
Requires: $\mathbf{P} \cup \mathbf{VC} \cup \mathbf{BBM} \subseteq \mathbf{P}' \wedge \mathbf{BBM} \cap \mathbf{P} = \varnothing \wedge \mathbf{VC} \cap \mathbf{P} = \varnothing$

Initialize()

$\mathsf{pp}^*_{\mathsf{vc}}, \mathsf{pp}^*_{\mathsf{bb}}, \mathsf{tally}^*, \pi^*, \mathsf{res}^*, \mathsf{VK}^*[*], \mathsf{Dec}[*] \leftarrow \bot$
$\mathsf{Dec}^*[*], \mathsf{VerRec}[*], \mathsf{subTime}[*,*] \leftarrow \bot$
$\mathsf{V}^*, \mathsf{V}_1, \mathsf{V}_2, \mathsf{Com} \leftarrow \varnothing$
$\mathsf{done}_1, \mathsf{done}_2, \mathsf{done}_3, \mathsf{done}_4, \mathsf{done}_5 \leftarrow \bot$
$T_0 \leftarrow \mathsf{T}()$
for $i \in [1,2,3,5,6,8,9]$ **do** $T_i \leftarrow T_{i-1} + \Delta_i$
$T_4 \leftarrow T_2 + \Delta_4$; $T_7 \leftarrow T_5 + \Delta_7$

P.BBSetup() for $\mathsf{P} \in \mathbf{BBM}$

if $\mathsf{T}() < T_0$ **then return** $\bot$
if $T_0 \leq \mathsf{T}() < T_1$ **then**
  $\mathsf{Sim}(\mathsf{P.BBSetup})$
  return $\bot$
if $\mathsf{T}() = T_1$ **then**
  $\mathsf{pp}'_{\mathsf{bb}} \leftarrow \mathsf{Sim}(\mathsf{P.BBSetup})$
  if $\neg\mathsf{done}_1$ **then** $(\mathsf{pp}^*_{\mathsf{bb}}, \mathsf{done}_1) \leftarrow (\mathsf{pp}'_{\mathsf{bb}}, \top)$
  if $|\mathbf{BBM} \cap \mathbf{C}| \leq t_{\mathsf{BB}}$ **then**
    $\mathsf{San}(\mathsf{pp}^*_{\mathsf{bb}})$: $\mathsf{pp}^*_{\mathsf{bb}} \neq \bot$
  return $\mathsf{pp}^*_{\mathsf{bb}}$
if $T_1 < \mathsf{T}()$ **then return** $\mathsf{pp}^*_{\mathsf{bb}}$

P.VCSetup() for $\mathsf{P} \in \mathbf{VC}$

if $\mathsf{T}() < T_0$ **then return** $\bot$
if $T_0 \leq \mathsf{T}() < T_1$ **then**
  $\mathsf{Sim}(\mathsf{P.VCSetup})$
  return $\bot$
if $\mathsf{T}() = T_1$ **then**
  $\mathsf{pp}'_{\mathsf{vc}} \leftarrow \mathsf{Sim}(\mathsf{P.VCSetup})$
  if $\neg\mathsf{done}_2$ **then** $(\mathsf{pp}^*_{\mathsf{vc}}, \mathsf{done}_2) \leftarrow (\mathsf{pp}'_{\mathsf{vc}}, \top)$
  if $|\mathbf{VC} \cap \mathbf{C}| \leq t_{\mathsf{VC}}$ **then**
    $\mathsf{San}(\mathsf{pp}^*_{\mathsf{vc}})$: $\mathsf{pp}^*_{\mathsf{vc}} \neq \bot$
  return $\mathsf{pp}^*_{\mathsf{vc}}$
if $T_1 < \mathsf{T}()$ **then return** $\mathsf{pp}^*_{\mathsf{vc}}$

P.Vote(layer, x) for $\mathsf{P} \in \mathbf{P}$

if $\mathsf{T}() < T_2$ **then return** $\bot$
if $T_2 \leq \mathsf{T}() < T_3$ **then**
  check $\mathsf{pp}^*_{\mathsf{vc}} \neq \bot \wedge \mathsf{Val}(x)$
  if layer $= 1$ **then**
    if $|\mathbf{VC} \cap \mathbf{C}| \leq t_{\mathsf{VC}}$ **then**
      $\psi \leftarrow \mathsf{Sim}(\mathsf{P.Vote}, \mathsf{layer}, 1^{|x|})$
    if $|\mathbf{VC} \cap \mathbf{C}| > t_{\mathsf{VC}}$ **then**
      $\psi \leftarrow \mathsf{Sim}(\mathsf{P.Vote}, \mathsf{layer}, x)$
    $\mathsf{San}(\psi)$: $\psi \neq \bot \wedge \mathsf{Dec}[\psi] = \bot$
    if $\mathsf{T}() \leq T_3$ **then**
      if $(\mathsf{P}, x) \notin \mathsf{V}_1$ **then** $\mathsf{V}_1 \leftarrow \mathsf{V}_1 \cup \{(\mathsf{P}, x)\}$
    $\mathsf{Dec}[\psi] \leftarrow x$
    return ψ
  if layer $= 2$ **then**
    if $|\mathbf{BBM} \cap \mathbf{C}| \leq t_{\mathsf{BB}} \wedge \mathsf{T}() \leq T_3 - \Delta \wedge (\mathsf{P}, x) \notin V_2$ **then**
      $V_2 \leftarrow V_2 \cup \{(\mathsf{P}, x)\}$
      $\mathsf{subTime}[\mathsf{P}, x] \leftarrow \mathsf{T}()$
    if $|\mathbf{BBM} \cap \mathbf{C}| \leq t_{\mathsf{BB}} \wedge \mathsf{T}() - \mathsf{subTime}[\mathsf{P}, x] \geq \Delta$ **then**
      if $|\mathbf{VC} \cap \mathbf{C}| \leq t_{\mathsf{VC}}$ **then**
        $\psi \leftarrow \mathsf{Sim}(\mathsf{P.Vote}, \mathsf{layer})$
        $\mathsf{San}(\psi)$: $\psi \neq \bot \wedge \mathsf{Dec}[\psi] = \bot \wedge \mathsf{VerRec}[\psi] \neq 0$
      if $|\mathbf{VC} \cap \mathbf{C}| > t_{\mathsf{VC}}$ **then**
        $\psi \leftarrow \mathsf{Sim}(\mathsf{P.Vote}, \mathsf{layer}, x)$
    if $\psi = \bot \vee \mathsf{VerRec}[\psi] = 0$ **then return** $\bot$
    $\mathsf{VerRec}[\psi] \leftarrow 1$
    $\mathsf{Dec}[\psi] \leftarrow x$
    return ψ
if $T_3 < \mathsf{T}()$ **then**
  return $\bot$

P.FinalizeVotes() for $\mathsf{P} \in \mathbf{BBM}$

if $\mathsf{T}() < T_2$ **then return** $\bot$
if $T_2 \leq \mathsf{T}() \leq T_4$ **then**
  $\mathsf{Sim}(\mathsf{P.FinalizeVotes})$
  return $\bot$
if $\mathsf{T}() = T_4$ **then**
  check $(\mathsf{pp}^*_{\mathsf{bb}} \neq \bot)$
  $\mathsf{V}' \leftarrow \mathsf{Sim}(\mathsf{P.FinalizeVotes})$
  if $\neg\mathsf{done}_3$ **then** $(\mathsf{V}^*, \mathsf{done}_3) \leftarrow (\mathsf{V}', \top)$
  $\mathsf{San}(\mathsf{V}^*)$: $\mathsf{Val}^1_{\mathsf{BB}}(\mathsf{V}^*, \mathsf{V}_1)$
  if $|\mathbf{BBM} \cap \mathbf{C}| \leq t_{\mathsf{BB}}$ **then**
    $\mathsf{San}(\mathsf{V}^*)$: $\mathsf{Val}^2_{\mathsf{BB}}(\mathsf{V}^*, \mathsf{V}_1, \mathsf{V}_2)$
  return ok
if $T_4 < \mathsf{T}() \leq T_9$ **then**
  $\mathsf{Sim}(\mathsf{P.FinalizeVotes})$
  return ok
if $T_7 < \mathsf{T}()$ **then**
  return $\bot$

To Vote, users first encrypt their votes via $\mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}$ and then sign them. For layer-2 voting, they submit this data via $\mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}$, in response to which they obtain a receipt. For layer-1 voting, they submit the signed and encrypted vote via a mainchain transaction to GovSC. This is used, for example, when a dishonest $\mathbf{BBM}$ server refuses to provide a valid receipt. The contract checks the vote for correctness by verifying signatures, PKI certificates, and vote validity, and, if it verifies, stores it in memory.

An important component of our protocol is the finalization of the list of votes by the $\mathbf{BBM}$ under FinalizeVotes. In doing so, the $\mathbf{BBM}$ must ensure that layer-1 and layer-2 votes are appropriately combined. To this end, the $\mathbf{BBM}$ removes from the list of layer-2 votes all layer-1 votes whose user verification keys also appear on it. It also removes from layer-2 votes all those with invalid proofs. (Looking ahead, when a complaint is submitted to GovSC, it will not be recorded unless accompanied by a valid proof.) These (now disjoint) sets of votes are combined and their respective Merkle roots computed. The $\mathbf{BBM}$ also prepares the encrypted tally CT using the Eval interface of $\mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}$. It then attaches a succinct proof (SNARK) that this encrypted tally corresponds to a set of votes with the computed Merkle root and a distinct set of user verification keys, all of which are eligible to vote. The proof also ensures that all ciphertexts are valid, correctly signed, and have valid PKI certificates. We refer to Relation 22 for

$\mathcal{F}_{\mathsf{L2Gov}}$ (Continued)

P.CheckInc(Q, ψ) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{VC} \cup \mathbf{BBM}$

78: **if** $\mathsf{T}() < T_5$ **then return** $\perp$
79: **if** $T_5 \leq \mathsf{T}() \leq T_6$ **then**
80: $b \leftarrow \mathsf{Sim}(\mathsf{P.CheckInc}, \mathsf{Q}, \psi)$
81: **if** $\mathsf{VerInc}[(\mathsf{Q}, \psi)] \neq \perp$ **then**
82: **return** $\mathsf{VerInc}[(\mathsf{Q}, \psi)]$
83: $x \leftarrow \mathsf{Dec}[\psi]$
84: **if** $(\mathsf{Q}, x) \notin \mathsf{V}^*$: $b \leftarrow 0$
85: **if** $|\mathbf{BBM} \cap \mathbf{C}| \leq t_{\mathsf{BB}}$ **then**
86: $\mathsf{R}_3 \leftarrow \mathsf{R}_3 \cup \{(\mathsf{P}, \mathsf{Q}, \psi, \mathsf{T}())\}$
87: $\mathsf{t}_0 \leftarrow \min\{t' : (\mathsf{P}, \mathsf{Q}, \psi, t') \in \mathsf{R}_3\}$
88: **if** $\mathsf{T}() - \mathsf{t}_0 \geq \Delta \wedge (\mathsf{Q}, x) \in \mathsf{V}^*$ **then**
89: $\mathsf{San}(b)$: $\mathsf{VerInc}[\mathsf{Q}, \psi] \neq 0 \wedge b = 1$
90: $\mathsf{R}_3 \leftarrow \mathsf{R}_3 \setminus \{(\mathsf{P}, \mathsf{Q}, \psi, t') \in \mathsf{R}_3 : t' \geq \mathsf{t}_0\}$
91: $\mathsf{VerInc}[\mathsf{Q}, \psi] \leftarrow b$
92: **return** b
93: **if** $T_6 < \mathsf{T}()$ **then return** $\perp$

P.Complain(Q, ψ) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{VC} \cup \mathbf{BBM}$

94: **if** $\mathsf{T}() < T_5$ **then return** $\perp$
95: **if** $T_5 \leq \mathsf{T}() < T_6$ **then**
96: $\mathsf{Sim}(\mathsf{P.Complain}, \mathsf{Q}, \psi)$
97: **if** $\mathsf{VerRec}[\psi] = 1$ **then**
98: $\mathsf{Com} \leftarrow \mathsf{Com} \cup \{(\mathsf{Q}, \psi)\}$
99: **return** ok
100: **if** $T_6 < \mathsf{T}()$ **then return** $\perp$

P.Resolve(Q, ψ) for $\mathsf{P} \in \mathbf{BBM}$

101: **if** $\mathsf{T}() < T_5$ **then return** $\perp$
102: **if** $T_5 \leq \mathsf{T}() \leq T_7$ **then**
103: $\mathsf{Sim}(\mathsf{P.Resolve}, \mathsf{Q}, \psi)$
104: $x \leftarrow \mathsf{Dec}[\psi]$
105: **if** $|\mathbf{BBM} \cap \mathbf{C}| \leq t_{\mathsf{BB}} \wedge (\mathsf{Q}, x) \in \mathsf{V}^*$ **then**
106: $\mathsf{Com} \leftarrow \mathsf{Com} \setminus \{(\mathsf{Q}, \psi)\}$
107: **return** ok
108: **if** $T_7 < \mathsf{T}()$ **then return** $\perp$

P.Tally() for $\mathsf{P} \in \mathbf{VC}$

109: **if** $\mathsf{T}() < T_8$ **then return** $\perp$
110: **if** $T_8 \leq \mathsf{T}() < T_9$ **then**
111: $\mathsf{Sim}(\mathsf{P.Tally})$
112: **return** $\perp$
113: **if** $\mathsf{T}() = T_9$ **then**
114: **check** $(\mathsf{pp}^*_{\mathsf{vc}} \neq \perp) \wedge \mathsf{V}^* \neq \varnothing$
115: $\mathsf{tally}^* \leftarrow F(\mathsf{V}^*)$
116: $\mathsf{tally}' \leftarrow \mathsf{Sim}(\mathsf{P.Tally}, \mathsf{tally}^*)$
117: **if** $\neg\mathsf{done}_4$ **then** $(\mathsf{tally}, \mathsf{done}_4) \leftarrow (\mathsf{tally}', \top)$
118: **if** $\mathsf{tally} \neq \mathsf{tally}^*$ **then return** $\perp$
119: **return** tally^*
120: **if** $T_9 < \mathsf{T}()$ **then return** $\perp$

P.GetResult() for $\mathsf{P} \in \mathbf{P} \cup \mathbf{VC} \cup \mathbf{BBM}$

121: $\mathsf{res}' \leftarrow \mathsf{Sim}(\mathsf{P.GetResult})$
122: **if** $\neg\mathsf{done}_5$ **then** $(\mathsf{res}, \mathsf{done}_5) \leftarrow (\mathsf{res}', \top)$
123: **if** $\mathsf{Com} \neq \varnothing \vee \mathsf{res} \neq \mathsf{tally}^*$ **then return** $\perp$
124: **return** res

the full details of the circuit with respect to which the SNARK is computed. The proof, Merkle roots, and CT are then posted to GovSC. Computing the actual tally and a proof of its correctness is performed by the $\mathbf{VC}$ via the Decrypt interface of $\mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}$. The result is signed and submitted to GovSC.

A. Security of π_{L2Gov}

We now state and prove the security theorem for the L2Gov protocol π_{L2Gov}.

Theorem 5 (Theorem 2, formal version)**.** *Let* $\mathbf{P}$, $\mathbf{VC}$, $\mathbf{BBM}$, t_{VC}, $t_{\mathsf{BB}} := 0$, Val, $\mathsf{Val}^1_{\mathsf{BB}}$, $\mathsf{Val}^2_{\mathsf{BB}}$, F, Δ, Δ_1, Δ_2, …, Δ_9 *be a given set of parameters with* $\mathbf{VC}$ *a set of parties of size* n *and* $t_{\mathsf{VC}} < n/2$, $\mathbf{BBM} := \{\mathsf{S}\}$, *a designated identifier for a single* $\mathbf{BBM}$, $t_{\mathsf{BB}} := 0$, $\mathsf{Val}(m) := (m \in [0, b])$ *for some integer* b, $\mathsf{Val}^1_{\mathsf{BB}}(V, V_1) := (V = \mathsf{Clean}_1(V, V_1))$ *for a given relation* Clean_1 *efficiently computable by the function* $\mathsf{Cleanup}_1$, $\mathsf{Val}^2_{\mathsf{BB}}(V, V_1, V_2) := (V = \mathsf{Clean}_2(V, V_1, V_2))$ *for a given relation* Clean_2 *efficiently computable by the function* $\mathsf{Cleanup}_2$, *and* $F(M) := \sum_{m \in M} m$. *Suppose further that all delay parameters are positive. Let* π_{L2Gov} *be as described in Prots. 21 and 23, with relation* $\mathcal{R}_{\mathsf{BBM}}$ *defined in 22, for this set of parameters and with respect to hybrids* $\mathcal{G}_{\mathsf{Clock}}[\mathbf{P}'_1]$, $\mathcal{G}_{\mathsf{MC}}[\mathbf{P}'_2, \mathsf{Val}_{\mathsf{MC}}, \Delta_{\mathsf{lat}}, \Delta_{\mathsf{slack}}, \mathsf{Genesis}]$ *with* $\mathbf{P} \cup \mathbf{VC} \cup \mathbf{BBM} \subseteq \mathbf{P}'_1 \cap \mathbf{P}'_2$, $\mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}[\mathbf{P}']$, $\mathcal{F}^{\mathsf{gv}}_{\mathsf{Sig}}[\mathbf{P}', \mathbf{P}']$, $\mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}[\mathbf{P}, \mathbf{BBM}, t_{\mathsf{BB}}, \mathsf{Val}^1_{\mathsf{BB}}, \Delta, \Delta^{\mathsf{b}}_1, \Delta^{\mathsf{b}}_2, \Delta^{\mathsf{b}}_3, \Delta^{\mathsf{b}}_4]$, $\mathcal{F}^{\mathsf{gv}}_{\mathsf{iNIZK}}[\mathbf{BBM}, \mathcal{F}_{\mathcal{R}_{\mathsf{BBM}}}]$, $\mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}[\mathbf{P}, \mathbf{VC}, t_{\mathsf{VC}}, \mathsf{Val}, F, \Delta^{\mathsf{d}}_1, \Delta^{\mathsf{d}}_2, \Delta^{\mathsf{d}}_3]$. *Then* π_{L2Gov} *UC-realizes* $\mathcal{F}_{\mathsf{L2Gov}}$ *for parameters* $\mathbf{P}$, $\mathbf{VC}$, $\mathbf{BBM}$, t_{VC}, t_{BB}, Val, $\mathsf{Val}^1_{\mathsf{BB}}$, $\mathsf{Val}^2_{\mathsf{BB}}$, F, Δ, Δ_1, Δ_2, …, Δ_9 *as long as the* Δ*'s satisfy the following conditions:*

- $\Delta_1 \geq \max\{\Delta^{\mathsf{d}}_1, \Delta^{\mathsf{b}}_1\} + \Delta_{\mathsf{mc}}$, *where* $\Delta_{\mathsf{mc}} := \Delta_{\mathsf{lat}} + \Delta_{\mathsf{slack}}$
- $\Delta_2 \geq \Delta_{\mathsf{mc}}$ *and* $\Delta_1 + \Delta_2 \geq \Delta^{\mathsf{b}}_1 + \Delta^{\mathsf{b}}_3$
- $\Delta_3 \geq \Delta^{\mathsf{b}}_3$
- $\Delta_4 \geq \Delta^{\mathsf{b}}_4$
- $\Delta_5 \geq \Delta_{\mathsf{mc}}$
- $\Delta_6 > 0$
- $\Delta_7 \geq \Delta_6 + \Delta^{\mathsf{b}}_4 + \Delta_{\mathsf{mc}}$
- $\Delta_8 \geq \Delta_{\mathsf{mc}}$ *and* $\Delta_1 + \Delta_2 + \Delta_4 + \Delta_5 + \Delta_7 + \Delta_8 > \Delta^{\mathsf{d}}_1 + \Delta^{\mathsf{d}}_2$
- $\Delta_9 \geq \Delta^{\mathsf{d}}_3$

Proof. For any given real-world adversary $\mathcal{A}$, we give a simulator Sim such that no distinguisher can distinguish a $(\pi_{\mathsf{L2Gov}}, \mathcal{A})$ session from a $(\mathcal{F}_{\mathsf{L2Gov}}, \mathsf{Sim})$ session.

The simulator internally runs hybrids $\mathcal{F}^{\mathsf{gv}}_{\mathsf{Sig}}[\mathbf{P}', \mathbf{P}']$, $\mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}[\mathbf{P}']$, $\mathcal{F}^{\mathsf{gv}}_{\mathsf{iNIZK}}[\mathbf{BBM}, \mathcal{F}_{\mathcal{R}_{\mathsf{BBM}}}]$, $\mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}[\mathbf{P}, \mathbf{BBM}, t_{\mathsf{BB}}, \mathsf{Val}^1_{\mathsf{BB}}, \Delta, \Delta^{\mathsf{b}}_1, \Delta^{\mathsf{b}}_2, \Delta^{\mathsf{b}}_3, \Delta^{\mathsf{b}}_4]$, and $\mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}[\mathbf{P}, \mathbf{VC}, t_{\mathsf{VC}}, \mathsf{Val}, F, \Delta^{\mathsf{d}}_1, \Delta^{\mathsf{d}}_2, \Delta^{\mathsf{d}}_3]$, where the adversary $\mathcal{A}$ plays the role of the simulator within the hybrid functionalities. It also maintains internal versions of the lists (denoted, as before, by the subscript int) kept by all the above ideal functionalities. As before, since most of the simulated functionalities have global verification interfaces, the environment queries the simulator on additional interfaces. The faithful simulation helps it return consistent responses

Protocol 21: $\pi_{\mathsf{L2Gov}}[\mathbf{P}, \mathbf{VC}, \mathbf{BBM}, t_{\mathsf{VC}}, t_{\mathsf{BB}}, \mathsf{Val}, \mathsf{Val}^1_{\mathsf{BB}}, \mathsf{Val}^2_{\mathsf{BB}}, \mathsf{Cleanup}, F, \Delta, \Delta_1, \ldots, \Delta_9, \mathsf{eID}]$

Parameters

1: Hybrids: $\mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}[\mathbf{P}, \mathbf{VC}, t_{\mathsf{VC}}, \mathsf{Val}, F, \Delta^d_1, \Delta^d_2, \Delta^d_3]$, $\mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}[\mathbf{P}, \mathbf{BBM}, t_{\mathsf{BB}}, \mathsf{Val}^1_{\mathsf{BB}}, \Delta, \Delta^b_1, \Delta^b_2, \Delta^b_3, \Delta^b_4]$, $\mathcal{F}^{\mathsf{gv}}_{\mathsf{iNIZK}}[\mathbf{BBM}, \mathcal{F}_{\mathcal{R}_{\mathsf{BBM}}}]$, $\mathcal{F}^{\mathsf{gv}}_{\mathsf{Sig}}[\mathbf{P}', \mathbf{P}']$, $\mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}[\mathbf{P}']$, $\mathcal{G}_{\mathsf{Clock}}[\mathbf{P}'_1]$, $\mathcal{G}_{\mathsf{MC}}[\mathbf{P}'_2, \mathsf{Val}_{\mathsf{MC}}, \Delta_{\mathsf{lat}}, \Delta_{\mathsf{slack}}, \mathsf{Genesis}]$

2: Requires: $\mathbf{P}' = \mathbf{P} \cup \mathbf{VC} \cup \mathbf{BBM} \wedge \mathbf{P}' \subseteq \mathbf{P}'_1 \cap \mathbf{P}'_2 \wedge \mathbf{BBM} \cap \mathbf{P} = \varnothing \wedge \mathbf{VC} \cap \mathbf{P} = \varnothing$

Initialize()

3: $T_0 \leftarrow \mathsf{T}()$

4: **for** $i \in [1, 2, 3, 5, 6, 8, 9]$ **do** $T_i \leftarrow T_{i-1} + \Delta_i$

5: $T_4 \leftarrow T_2 + \Delta_4$; $T_7 \leftarrow T_5 + \Delta_7$

S.BBSetup() for $\mathsf{S} \in \mathbf{BBM} = \{\mathsf{S}\}$

6: **if** $\mathsf{T}() < T_0$ **then return** $\perp$

7: **if** $T_0 \leq \mathsf{T}() < T_1$ **then**

8: $(\mathsf{code}, \mathsf{mem}, *) \leftarrow \mathsf{S.ReadConfig}(\mathsf{eID})$

9: **if** $\mathsf{code} = \perp$ **then** $\mathsf{S.Init}(\mathsf{eID})$

10: **check** $\mathsf{S.checkSC}(\mathsf{eID})$

11: $\pi_{\mathsf{mr}} \leftarrow \mathsf{MerkleProof}(\mathsf{S}, \mathbf{BBM})$

12: **if** $\mathsf{vk} = \perp$ **then**

13: $\mathsf{vk} \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{Sig}}.\mathsf{S.Gen}()$

14: $\mathsf{pp}^* \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}.\mathsf{S.Setup}()$

15: $\mathsf{cert} \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}.\mathsf{S.Register}(\mathsf{pp}^*, \mathsf{vk})$

16: $\mathsf{pp}_{\mathsf{bb}} \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}.\mathsf{S.Setup}()$

17: $\sigma \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{Sig}}.\mathsf{S.Sign}(\mathsf{eID}, \text{“setup-bb”}, \mathsf{pp}_{\mathsf{bb}})$

18: $\mathcal{G}_{\mathsf{MC}}.\mathsf{S.Submit}(\mathsf{eID}, \text{“setup-bb”}, \mathsf{S}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, \mathsf{pp}_{\mathsf{bb}}, \sigma)$

19: **return** $\perp$

20: **if** $T_1 \leq \mathsf{T}()$ **then return** $\mathsf{pp}_{\mathsf{bb}}$

P.VCSetup() for $\mathsf{P} \in \mathbf{VC}$

21: **if** $\mathsf{T}() < T_0$ **then return** $\perp$

22: **if** $T_0 \leq \mathsf{T}() < T_1$ **then**

23: $(\mathsf{code}, \mathsf{mem}, *) \leftarrow \mathsf{P.ReadConfig}(\mathsf{eID})$

24: **if** $\mathsf{code} = \perp$ **then** $\mathsf{P.Init}(\mathsf{eID})$

25: **check** $\mathsf{P.checkSC}(\mathsf{eID})$

26: $\pi_{\mathsf{mr}} \leftarrow \mathsf{MerkleProof}(\mathsf{P}, \mathbf{VC})$

27: **if** $\mathsf{vk} = \perp$ **then**

28: $\mathsf{vk} \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{Sig}}.\mathsf{P.Gen}()$

29: $\mathsf{pp}^* \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}.\mathsf{P.Setup}()$

30: $\mathsf{cert} \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}.\mathsf{P.Register}(\mathsf{pp}^*, \mathsf{vk})$

31: $\mathsf{pp}_{\mathsf{vc}} \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}.\mathsf{P.Setup}()$

32: $\sigma \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{Sig}}.\mathsf{P.Sign}(\mathsf{eID}, \text{“setup-vc”}, \mathsf{pp}_{\mathsf{vc}})$

33: $\mathcal{G}_{\mathsf{MC}}.\mathsf{P.Submit}(\mathsf{eID}, \text{“setup-vc”}, \mathsf{P}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, \mathsf{pp}_{\mathsf{vc}}, \sigma)$

34: **return** $\perp$

35: **if** $T_1 \leq \mathsf{T}()$ **then return** $\mathsf{pp}_{\mathsf{vc}}$

P.Vote(layer, x) for $\mathsf{P} \in \mathbf{P}$

36: **if** $\mathsf{T}() < T_2$ **then return** $\perp$

37: **if** $T_2 \leq \mathsf{T}() \leq T_3$ **then**

38: **if** $\mathsf{vk} = \perp$ **then**

39: $\mathsf{vk} \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{Sig}}.\mathsf{P.Gen}()$

40: $\mathsf{pp}^* \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}.\mathsf{P.Setup}()$

41: $\mathsf{cert} \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}.\mathsf{P.Register}(\mathsf{pp}^*, \mathsf{vk})$

42: **check** $\mathsf{P.checkSC}(\mathsf{eID})$

43: $(_, \mathsf{mem}, _) \leftarrow \mathsf{P.ReadConfig}(\mathsf{eID})$

44: $\mathsf{pp}_{\mathsf{vc}} \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“fin-pp-vc”}, *)$

45: $\pi_{\mathsf{mr}} \leftarrow \mathsf{MerkleProof}(\mathsf{P}, \mathbf{P})$

46: $c \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}.\mathsf{P.Encrypt}(\mathsf{pp}_{\mathsf{vc}}, x)$

47: **if** $\mathsf{layer} = 1$ **then**

48: $\sigma \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{Sig}}.\mathsf{P.Sign}(\mathsf{eID}, \text{“L1-vote”}, c)$

49: $\rho \leftarrow \epsilon$

50: $\mathcal{G}_{\mathsf{MC}}.\mathsf{P.Submit}(\mathsf{eID}, \text{“L1-vote”}, \mathsf{P}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, c, \sigma)$

51: **if** $\mathsf{layer} = 2$ **then**

52: $\sigma \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{Sig}}.\mathsf{P.Sign}(\mathsf{eID}, \text{“L2-vote”}, c)$

53: $\rho \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}.\mathsf{P.Submit}(\mathsf{eID}, \text{“L2-vote”}, \mathsf{P}, \mathsf{vk}, \mathsf{cert}, c, \sigma)$

54: **return** (c, ρ)

55: **if** $T_3 < \mathsf{T}()$ **then return** $\perp$

S.FinalizeVotes() for $\mathsf{P} \in \mathbf{BBM} = \{\mathsf{S}\}$

56: **if** $\mathsf{T}() < T_2$ **then return** $\perp$

57: **if** $T_2 \leq \mathsf{T}() < T_4$ **then**

58: $\mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}.\mathsf{S.FormBB}()$

59: **return** $\perp$

60: **if** $\mathsf{T}() = T_4$ **then**

61: $(\mathsf{code}, \mathsf{mem}, _) \leftarrow \mathsf{S.ReadConfig}(\mathsf{eID})$

62: **check** code

63: **check** $\mathsf{S.checkSC}(\mathsf{eID})$

64: $\mathsf{pp}_{\mathsf{vc}} \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“fin-pp-vc”}, *)$

65: $\mathsf{BB}_1 \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“L1-vote”}, *)$

66: $\mathsf{BB}_1 \leftarrow \mathsf{Cleanup}(\mathsf{BB}_1)$

67: $\mu_1 \leftarrow \mathsf{MerkleRoot}(\mathsf{BB}_1)$

68: $(\mathsf{BB}_2, \mu_2) \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}.\mathsf{S.FormBB}()$

69: $\mathsf{BB}_2 \leftarrow \mathsf{BB}_2 \setminus \{x \in \mathsf{BB}_2 : x.\mathsf{vk} \in \mathsf{BB}_1.\mathsf{vk}\}$

70: $\mathsf{BB}_2 \leftarrow \mathsf{BB}_2 \setminus \{x \in \mathsf{BB}_2 : \neg \mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}.\mathsf{S.VerifyEnc}(\mathsf{pp}_{\mathsf{vc}}, x.c)\}$

71: $v_{\mathsf{voter}} \leftarrow \mathsf{MerkleRoot}(\mathbf{P})$

72: $\pi_{\mathsf{mr}} \leftarrow \mathsf{MerkleProof}(\mathsf{S}, \mathbf{BBM})$

73: $\mathsf{CT} \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}.\mathsf{S.Eval}(\mathsf{pp}_{\mathsf{vc}}, \mathsf{BB}_1 \cup \mathsf{BB}_2)$

74: $\mathsf{pp}^* \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}.\mathsf{S.Setup}()$

75: $x \leftarrow (\mathsf{pp}^*, \mathsf{pp}_{\mathsf{vc}}, \mathsf{CT}, \mu_1, \mu_2, v_{\mathsf{voter}})$

76: $w \leftarrow (\mathsf{BB}_1, \mathsf{BB}_2, \mathbf{P})$

77: $\pi_{\mathsf{bb}} \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{iNIZK}}[\mathbf{BBM}, \mathcal{F}_{\mathcal{R}_{\mathsf{BBM}}}].\mathsf{S.Prove}(x, w)$

78: $\sigma \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{Sig}}.\mathsf{S.Sign}(\mathsf{eID}, \text{“collect”}, \mathsf{CT}, \mu_1, \mu_2, v_{\mathsf{voter}}, \pi_{\mathsf{bb}})$

79: $\mathcal{G}_{\mathsf{MC}}.\mathsf{S.Submit}(\mathsf{eID}, \text{“collect”}, \mathsf{S}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, \mathsf{CT}, \mu_1, \mu_2, v_{\mathsf{voter}}, \pi_{\mathsf{bb}}, \sigma)$

80: **if** $T_4 < \mathsf{T}() \leq T_9$ **then**

81: $\mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}.\mathsf{S.FormBB}()$

82: **return** $\perp$

83: **if** $T_7 < \mathsf{T}()$ **then**

84: **return** $\perp$

even when the adversary responds inconsistently or incorrectly.

On being invoked in any of the interfaces below, the simulator will essentially follow the code of π_{L2Gov} except when it lacks the required ‘secret’, or must output something ‘extra’ over what is computed in π_{L2Gov}. We will argue why we can omit code for output sanitization or consistency checks in several interfaces of $\mathcal{F}_{\mathsf{L2Gov}}$, thereby making the ideal-world code indistinguishable from the π_{L2Gov} code. We now detail the simulator for $\mathcal{F}_{\mathsf{L2Gov}}$:

$\mathcal{F}_{\mathsf{L2Gov}}.\mathsf{P.BBSetup}()$:

Sim(P.BBSetup): The simulator follows the π_{L2Gov} code. In particular, it checks (using checkSC) if the correct voting smart contract exists on $\mathcal{G}_{\mathsf{MC}}$ (initiating the contract if not present), creates a MerkleProof π_{mr} of its identity S being in the $\mathbf{BBM}$ list, generates parameters $\mathsf{pp}_{\mathsf{bb}}$ using $\mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}.\mathsf{P.Setup}$, signs it ($\sigma$) using a fresh certified key vk, and submits π_{mr}, $\mathsf{pp}_{\mathsf{bb}}$, and σ to the smart contract.

$\mathsf{pp}'_{\mathsf{bb}} \leftarrow \mathsf{Sim}(\mathsf{P.BBSetup})$: The simulator returns $\mathsf{pp}_{\mathsf{bb}}$. As $\mathsf{pp}_{\mathsf{bb}}$ is generated using $\mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}$, the sanitization in lines 17-18 (functionality 18) makes the sanitization of lines 17-18 in functionality 20 redundant. Thus, the code of $\mathcal{F}_{\mathsf{L2Gov}}$

π_{L2Gov} (Continued)

P.CheckInc(Q, ψ) for $\mathsf{P} \in \mathbf{P}$

85: **if** $\mathsf{T}() < T_5$ **then return** $\perp$
86: **if** $T_5 \leq \mathsf{T}() < T_6$ **then**
87: **check** P.checkSC(eID)
88: $(_, \mathsf{mem}, _) \leftarrow$ P.ReadConfig(eID)
89: $\mathsf{BB}_1 \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“L1-vote”}, *)$
90: **if** $(\mathsf{Q}, \psi) \in \mathsf{BB}_1$ **then return** 1
91: $(_, _, \mu_2) \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“fin-root”}, *)$
92: $(c, \rho) \leftarrow \mathsf{parse}(\psi)$
93: $\pi \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}$.P.GetIncProof$(\mu_2, \mathsf{Q}, c)$
94: **return** $\mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}$.P.VerifyIncProof$(\mu_2, (\mathsf{Q}, c), \pi)$
95: **if** $T_6 < \mathsf{T}()$ **then return** $\perp$

P.Complain(Q, ψ) for $\mathsf{P} \in \mathbf{P} \cup \mathbf{VC} \cup \mathbf{BBM}$

96: **if** $\mathsf{T}() < T_5$ **then return** $\perp$
97: **if** $T_5 \leq \mathsf{T}() < T_6$ **then**
98: $\mathcal{G}_{\mathsf{MC}}$.P.Submit(eID, “complain”, Q, ψ)
99: **if** $T_6 < \mathsf{T}()$ **then return** $\perp$

S.Resolve(Q, ψ) for $\mathsf{S} \in \mathbf{BBM} = \{\mathsf{S}\}$

100: **if** $\mathsf{T}() < T_5$ **then return** $\perp$
101: **if** $T_5 \leq \mathsf{T}() < T_7$ **then**
102: **check** P.checkSC(eID)
103: $(_, \mathsf{mem}, _) \leftarrow$ S.ReadConfig(eID)
104: $\pi_{\mathsf{mr}} \leftarrow \mathsf{MerkleProof}(\mathsf{S}, \mathbf{BBM})$
105: $(_, \mu_1, \mu_2) \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“fin-root”}, *)$
106: $(c, \rho) \leftarrow \mathsf{parse}(\psi)$
107: $\pi \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}$.S.GetIncProof$(\mu_2, \mathsf{Q}, c)$
108: $\sigma \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{Sig}}$.S.Sign(eID, “resolve”, Q, ψ, π)
109: $\mathcal{G}_{\mathsf{MC}}$.S.Submit(eID, “resolve”, S, vk, cert, $\pi_{\mathsf{mr}}, \mathsf{Q}, \psi, \pi, \sigma$)
110: **if** $T_7 < \mathsf{T}()$ **then return** $\perp$

P.Tally() for $\mathsf{P} \in \mathbf{VC}$

111: **if** $\mathsf{T}() < T_8$ **then return** $\perp$
112: **if** $T_8 \leq \mathsf{T}() \leq T_9$ **then**
113: **check** P.checkSC(eID)
114: $(_, \mathsf{mem}, _) \leftarrow$ P.ReadConfig(eID)
115: $(v_{\mathsf{voter}}, v_{\mathsf{vc}}, v_{\mathsf{bb}}, \mathsf{pp}^*) \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“init”}, *, *, *)$
116: $(\mathsf{CT}, \mu_1, \mu_2) \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“fin-root”}, *)$
117: $(\mathsf{tally}, \pi_{\mathsf{tally}}) \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}$.P.Decrypt(CT)
118: $\pi_{\mathsf{mr}} \leftarrow \mathsf{MerkleProof}(\mathsf{P}, \mathbf{VC})$
119: $\sigma \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{Sig}}$.P.Sign(eID, “tally”, tally, π_{tally})
120: $\mathcal{G}_{\mathsf{MC}}$.S.Submit(eID, “tally”, P, vk, cert, π_{mr}, CT, tally, $\pi_{\mathsf{tally}}, \sigma$)
121: **return** $(\mathsf{tally}, \pi_{\mathsf{tally}})$
122: **if** $T_9 < \mathsf{T}()$ **then return** $\perp$

P.GetResult() for $\mathsf{P} \in \mathbf{P} \cup \mathbf{VC} \cup \mathbf{BBM}$

123: **check** P.checkSC(eID) // pp^* is checked to be correct
124: $(_, \mathsf{mem}, _) \leftarrow$ P.ReadConfig(eID)
125: $\mathsf{res} \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“fin-tally”}, *)$
126: **return** res

Proc. P.Init(eID)

127: $v_{\mathsf{voter}} \leftarrow \mathsf{MerkleRoot}(\mathbf{P})$
128: $v_{\mathsf{vc}} \leftarrow \mathsf{MerkleRoot}(\mathbf{VC})$
129: $v_{\mathsf{bb}} \leftarrow \mathsf{MerkleRoot}(\mathbf{BBM})$
130: $\mathsf{pp}^* \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}$.P.Setup()
131: $\mathcal{G}_{\mathsf{MC}}$.P.Submit(eID, GovSC, “init”, $v_{\mathsf{voter}}, v_{\mathsf{vc}}, v_{\mathsf{bb}}, \mathsf{pp}^*$, state := bb-setup)

Proc. P.ReadConfig(eID)

132: $\log \leftarrow \mathcal{G}_{\mathsf{MC}}$.P.Read()
133: **return** ContractState(Genesis, log)[eID]

Proc. P.checkSC(eID)

134: $v_{\mathsf{voter}} \leftarrow \mathsf{MerkleRoot}(\mathbf{P})$
135: $v_{\mathsf{vc}} \leftarrow \mathsf{MerkleRoot}(\mathbf{VC})$
136: $v_{\mathsf{bb}} \leftarrow \mathsf{MerkleRoot}(\mathbf{BBM})$
137: $\mathsf{pp}^* \leftarrow \mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}$.P.Setup()
138: $\log \leftarrow \mathcal{G}_{\mathsf{MC}}$.P.Read()
139: $(\mathsf{code}, \mathsf{mem}, *) \leftarrow \mathsf{ContractState}(\mathsf{Genesis}, \log)[\mathsf{eID}]$
140: $(v'_{\mathsf{voter}}, v'_{\mathsf{vc}}, v'_{\mathsf{bb}}, \mathsf{pp}') \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“init”}, *)$
141: **return** $(\mathsf{code}, v'_{\mathsf{voter}}, v'_{\mathsf{vc}}, v'_{\mathsf{bb}}, \mathsf{pp}') = (\mathsf{GovSC}, v_{\mathsf{voter}}, v_{\mathsf{vc}}, v_{\mathsf{bb}}, \mathsf{pp}^*)$

Relation 22: $\mathcal{R}_{\mathsf{BBM}}$

$\mathcal{R}_{\mathsf{BBM}}((\mathsf{pp}^*, \mathsf{pp}_{\mathsf{vc}}, \mathsf{CT}, \mu_1, \mu_2, v_{\mathsf{voter}}), (\mathsf{BB}_1, \mathsf{BB}_2, \mathbf{P}))$

1: **check** $\mathsf{MerkleRoot}(\mathsf{BB}_1) = \mu_1$ // BB_1 correctness is enforced by GovSC
2: **check** $\mathcal{F}^{\mathsf{gv}}_{\mathsf{BB}}$.P.Com$(\mathsf{BB}_2) = \mu_2$
3: **check** $\mathsf{MerkleRoot}(\mathbf{P}) = v_{\mathsf{voter}}$
4: **check** $(\mathsf{BB}_1 \cup \mathsf{BB}_2).\mathsf{P} \subseteq \mathbf{P}$
5: **check** $(\mathsf{BB}_1 \cup \mathsf{BB}_2).\mathsf{vk}$ are distinct
6: **check** $\mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}$.P.Eval$(\mathsf{pp}_{\mathsf{vc}}, \mathsf{BB}_1 \cup \mathsf{BB}_2) = \mathsf{CT}$
7: **for** $(\mathsf{P}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, c, \sigma) \in \mathsf{BB}_2$ **do**
8: **check** $\mathcal{F}^{\mathsf{gv}}_{\mathsf{PKI}}$.P.certVerify$(\mathsf{pp}^*, \mathsf{P}, \mathsf{vk}, \mathsf{cert})$
9: **check** $\mathcal{F}^{\mathsf{gv}}_{\mathsf{Sig}}$.P.Verify$(\mathsf{vk}, (\mathsf{eID}, \text{“L2-vote”}, c), \sigma)$
10: **check** $\mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}$.P.VerifyEnc$(\mathsf{pp}_{\mathsf{vc}}, c)$

and π_{L2Gov} for the interface BBSetup is identical.

$\mathcal{F}_{\mathsf{L2Gov}}$.P.VCSetup():

Sim(P.VCSetup): The simulator follows the π_{L2Gov} code. In particular, it checks (using checkSC) that the correct voting smart contract exists on $\mathcal{G}_{\mathsf{MC}}$ (initiating the contract if not present), creates a MerkleProof π_{mr} of its identity P being in the $\mathbf{VC}$ list, generates parameters $\mathsf{pp}_{\mathsf{vc}}$ using $\mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}$.P.Setup, signs it ($\sigma$) using a fresh certified key vk, and submits π_{mr}, $\mathsf{pp}_{\mathsf{vc}}$, and σ to the smart contract.

$\mathsf{pp}'_{\mathsf{vc}} \leftarrow$ Sim(P.VCSetup): The simulator returns $\mathsf{pp}_{\mathsf{vc}}$. As $\mathsf{pp}_{\mathsf{vc}}$ is generated using $\mathcal{F}^{\mathsf{gv}}_{\mathsf{DHE}}$, the sanitization in functionality 16, lines 15-16, makes the sanitization of lines 28-29 in functionality 20 redundant. Thus, the code of $\mathcal{F}_{\mathsf{L2Gov}}$

Protocol 23: $\mathsf{GovSC}[t_{\mathsf{BB}} := 0, n := |\mathbf{VC}|, t_{\mathsf{VC}}]$

Initialize()
1: $\mathsf{res} \leftarrow 0; \mathsf{funds} \leftarrow 0; \mathsf{state} \leftarrow$ bb-setup

setup-bb($\mathsf{P}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, \mathsf{pp}_{\mathsf{bb}}, \sigma$)
2: **check** state = bb-setup
3: **check** $T_0 \le \mathsf{T}() \le T_1 - \Delta_1$
4: $(v_{\mathsf{voter}}, v_{\mathsf{vc}}, v_{\mathsf{bb}}, \mathsf{pp}^*) \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“init”}, *)$
5: **check** $\mathsf{MerkleVerify}(v_{\mathsf{bb}}, \mathsf{P}, \pi_{\mathsf{mr}})$
6: **check** $\mathcal{F}_{\mathsf{PKI}}^{\mathsf{gv}}.\mathsf{P.certVerify}(\mathsf{pp}^*, \mathsf{P}, \mathsf{vk}, \mathsf{cert})$
7: **check** $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}.\mathsf{P.Verify}(\mathsf{vk}, (\mathsf{eID}, \text{“setup-bb}''\text{”}, \mathsf{pp}_{\mathsf{bb}}), \sigma)$
8: $\mathsf{mem} \leftarrow \mathsf{mem} : (\text{“pp-bb”}, \mathsf{vk}, \mathsf{pp}_{\mathsf{bb}})$
9: **check** $|\{\mathsf{vk}' : (\text{“pp-bb”}, \mathsf{vk}', \mathsf{pp}_{\mathsf{bb}}) \in \mathsf{mem}\}| > t_{\mathsf{BB}}$
10: **check** $(\text{“fin-pp-bb”}, *) \notin \mathsf{mem}$
11: $\mathsf{mem} \leftarrow \mathsf{mem} : (\text{“fin-pp-bb”}, \mathsf{pp}_{\mathsf{bb}})$
12: $\mathsf{state} \leftarrow$ vc-setup

setup-vc($\mathsf{P}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, \mathsf{pp}_{\mathsf{vc}}, \sigma$)
13: **check** state = vc-setup
14: **check** $T_0 \le \mathsf{T}() \le T_1 - \Delta_1$
15: $(v_{\mathsf{voter}}, v_{\mathsf{vc}}, v_{\mathsf{bb}}, \mathsf{pp}^*) \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“init”}, *)$
16: **check** $\mathsf{MerkleVerify}(v_{\mathsf{vc}}, \mathsf{P}, \pi_{\mathsf{mr}})$
17: **check** $\mathcal{F}_{\mathsf{PKI}}^{\mathsf{gv}}.\mathsf{P.certVerify}(\mathsf{pp}^*, \mathsf{P}, \mathsf{vk}, \mathsf{cert})$
18: **check** $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}.\mathsf{P.Verify}(\mathsf{vk}, (\mathsf{eID}, \text{“setup-vc}''\text{”}, \mathsf{pp}_{\mathsf{vc}}), \sigma)$
19: $\mathsf{mem} \leftarrow \mathsf{mem} : (\text{“pp-vc”}, \mathsf{vk}, \mathsf{pp}_{\mathsf{vc}})$
20: **check** $|\{\mathsf{vk}' : (\text{“pp-vc”}, \mathsf{vk}', \mathsf{pp}_{\mathsf{vc}}) \in \mathsf{mem}\}| > n - t_{\mathsf{VC}}$
21: **check** $(\text{“fin-pp-vc”}, *) \notin \mathsf{mem}$
22: $\mathsf{mem} \leftarrow \mathsf{mem} : (\text{“fin-pp-vc”}, \mathsf{pp}_{\mathsf{vc}})$
23: $\mathsf{state} \leftarrow$ voting

L1-vote($\mathsf{P}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, c, \sigma$)
24: **check** state = voting
25: **check** $T_2 \le \mathsf{T}() \le T_3 - \Delta_1$
26: $(v_{\mathsf{voter}}, v_{\mathsf{vc}}, v_{\mathsf{bb}}, \mathsf{pp}^*) \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“init”}, *)$
27: $\mathsf{pp}_{\mathsf{vc}} \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“fin-pp-vc”}, *)$
28: **check** $\mathsf{MerkleVerify}(v_{\mathsf{voter}}, \mathsf{P}, \pi_{\mathsf{mr}})$
29: **check** $\mathcal{F}_{\mathsf{PKI}}^{\mathsf{gv}}.\mathsf{P.certVerify}(\mathsf{pp}^*, \mathsf{P}, \mathsf{vk}, \mathsf{cert})$
30: **check** $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}.\mathsf{P.Verify}(\mathsf{vk}, (\mathsf{eID}, \text{“L1-vote”}, c), \sigma)$
31: **check** $(\text{“L1-vote”}, \mathsf{P}, *) \notin \mathsf{mem}$
32: **check** $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}.\mathsf{P.VerifyEnc}(\mathsf{pp}_{\mathsf{vc}}, c)$
33: $\mathsf{mem} \leftarrow \mathsf{mem} : (\text{“L1-vote”}, \mathsf{P}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, c, \sigma)$

collect($\mathsf{S}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, \mathsf{CT}, \mu_1, \mu_2, v_{\mathsf{voter}}, \pi_{\mathsf{bb}}, \sigma$)
34: **check** state = voting
35: **check** $T_2 \le \mathsf{T}() \le T_4 - \Delta_1$
36: $(v_{\mathsf{voter}}, v_{\mathsf{vc}}, v_{\mathsf{bb}}, , \mathsf{pp}^*) \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“init”}, *)$
37: $\mathsf{pp}_{\mathsf{vc}} \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“fin-pp-vc”}, *)$
38: **check** $\mathsf{MerkleVerify}(v_{\mathsf{bb}}, \mathsf{S}, \pi_{\mathsf{mr}})$
39: **check** $\mathcal{F}_{\mathsf{PKI}}^{\mathsf{gv}}.\mathsf{P.certVerify}(\mathsf{pp}^*, \mathsf{S}, \mathsf{vk}, \mathsf{cert})$
40: $\mathsf{BB}_1 \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“L1-vote”}, *)$
41: $\mathsf{BB}_1 \leftarrow \mathsf{Cleanup}(\mathsf{BB}_1)$
42: **check** $\mathsf{MerkleRoot}(\mathsf{BB}_1) = \mu_1$
43: **check** $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}.\mathsf{P.Verify}(\mathsf{vk}, (\mathsf{eID}, \text{“collect”}, \mathsf{CT}, \mu_1, \mu_2, v_{\mathsf{voter}}, \pi_{\mathsf{bb}}), \sigma)$
44: **check** $\mathcal{F}_{\mathsf{iNIZK}}^{\mathsf{gv}}[\mathbf{BBM}, \mathcal{F}_{\mathcal{R}_{\mathsf{BBM}}}].\mathsf{P.Verify}(\mathsf{pp}^*, \mathsf{pp}_{\mathsf{vc}}, \mathsf{CT}, \mu_1, \mu_2, v_{\mathsf{voter}}, \pi_{\mathsf{bb}})$
45: $\mathsf{mem} \leftarrow \mathsf{mem} : (\text{“fin-root”}, \mathsf{CT}, \mu_1, \mu_2)$
46: $\mathsf{state} \leftarrow$ dispute

complain(Q, ψ)
47: **check** state = dispute
48: **check** $T_5 \le \mathsf{T}() \le T_6 - \Delta_1$
49: $\mathsf{pp}_{\mathsf{bb}} \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“fin-pp-bb”}, *)$
50: $\mathsf{pp}_{\mathsf{vc}} \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“fin-pp-vc”}, *)$
51: $(c', \rho) \leftarrow \mathsf{parse}(\psi)$
52: **check** $\mathcal{F}_{\mathsf{BB}}^{\mathsf{gv}}.\mathsf{P.VerifyRec}(\mathsf{pp}_{\mathsf{bb}}, c', \rho)$
53: **check** $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}.\mathsf{P.VerifyEnc}(\mathsf{pp}_{\mathsf{vc}}, c')$
54: $\mathsf{BB}_1 \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“L1-vote”}, *)$
55: **check** $c' \notin \mathsf{BB}_1$
56: $\mathsf{mem} \leftarrow \mathsf{mem} : (\text{“complain”}, (\mathsf{Q}, \psi))$

resolve($\mathsf{S}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, \mathsf{Q}, \psi, \pi, \sigma$)
57: **check** state = dispute
58: **if** $\mathsf{T}() < T_7$ **then** $\mathsf{state} \leftarrow$ tallying
59: **check** $T_5 \le \mathsf{T}() \le T_7 - \Delta_1$
60: $(v_{\mathsf{voter}}, v_{\mathsf{vc}}, v_{\mathsf{bb}}, \mathsf{pp}^*) \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“init”}, *)$
61: **check** $\mathsf{MerkleVerify}(v_{\mathsf{bb}}, \mathsf{S}, \pi_{\mathsf{mr}})$
62: **check** $\mathcal{F}_{\mathsf{PKI}}^{\mathsf{gv}}.\mathsf{P.certVerify}(\mathsf{pp}^*, \mathsf{S}, \mathsf{vk}, \mathsf{cert})$
63: $(_, _, \mu_2) \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“fin-root”}, *)$
64: **check** $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}.\mathsf{P.Verify}(\mathsf{vk}, (\mathsf{eID}, \text{“resolve”}, \mathsf{Q}, \psi, \pi), \sigma)$
65: $(c', \rho) \leftarrow \mathsf{parse}(\psi)$
66: **check** $\mathcal{F}_{\mathsf{BB}}^{\mathsf{gv}}.\mathsf{P.VerifyIncProof}(\mu_2, (\mathsf{Q}, c'), \pi)$
67: $\mathsf{mem} \leftarrow \mathsf{mem} \setminus (\text{“complain”}, (\mathsf{Q}, \psi))$

tally($\mathsf{P}, \mathsf{vk}, \mathsf{cert}, \pi_{\mathsf{mr}}, \mathsf{tally}, \pi_{\mathsf{tally}}, \sigma$)
68: **check** state = tallying
69: **check** $T_8 \le \mathsf{T}() \le T_9 - \Delta_1$
70: $(v_{\mathsf{voter}}, v_{\mathsf{vc}}, v_{\mathsf{bb}}, \mathsf{pp}^*) \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“init”}, *)$
71: **check** $(\text{“complain”}, *) \notin \mathsf{mem}$
72: **check** $\mathsf{MerkleVerify}(v_{\mathsf{vc}}, \mathsf{P}, \pi_{\mathsf{mr}})$
73: **check** $\mathcal{F}_{\mathsf{PKI}}^{\mathsf{gv}}.\mathsf{P.certVerify}(\mathsf{pp}^*, \mathsf{P}, \mathsf{vk}, \mathsf{cert})$
74: **check** $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}.\mathsf{P.Verify}(\mathsf{vk}, (\mathsf{eID}, \text{“tally”}, \mathsf{tally}, \pi_{\mathsf{tally}}), \sigma)$
75: $\mathsf{pp}_{\mathsf{vc}} \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“fin-pp-vc”}, *)$
76: $(\mathsf{CT}, _, _) \leftarrow \mathsf{mem}|\mathrm{grep}(\text{“fin-root”}, *)$
77: **check** $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}.\mathsf{P.VerifyDec}(\mathsf{pp}_{\mathsf{vc}}, \mathsf{CT}, \mathsf{tally}, \pi_{\mathsf{tally}})$
78: $\mathsf{mem} \leftarrow \mathsf{mem} : (\text{“fin-tally”}, \mathsf{tally})$
79: $\mathsf{state} \leftarrow$ Result

and π_{L2Gov} for the interface VCSetup is identical.

$\mathcal{F}_{\mathsf{L2Gov}}.\mathsf{P.Vote}()$:

$\psi \leftarrow \mathsf{Sim}(\mathsf{P.Vote}, \mathsf{layer}, 1^{|x|})$: The simulator chooses an arbitrary (but valid) message msg of length $|x|$ and follows the π_{L2Gov} code, in particular computing c using $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}.\mathsf{P.Encrypt}(\mathsf{pp}_{\mathsf{vc}}, \mathsf{msg})$. Note that in this case the simulator of $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}$ gets only $|\mathsf{msg}|$ as input, and this is the same as in protocol π_{DHE}. Then, it signs c using vk, and if layer $= 1$, submits it to $\mathcal{G}_{\mathsf{MC}}$, else it sends it via $\mathcal{F}_{\mathsf{BB}}^{\mathsf{gv}}.\mathsf{P.Submit}$ to get back ρ. It returns (c, ρ) (note that $\rho = \perp$ when layer $= 1$).

$\psi \leftarrow \mathsf{Sim}(\mathsf{P.Vote}, \mathsf{layer}, x)$: The simulator follows the π_{L2Gov} code, in particular computing c using $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}.\mathsf{P.Encrypt}(\mathsf{pp}_{\mathsf{vc}}, x)$. Note that in this case the simulator of $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}$ gets x as input, and this is the same as in protocol π_{DHE}. Then, it signs c using vk, and if layer $= 1$, submits it to $\mathcal{G}_{\mathsf{MC}}$, else it sends it via $\mathcal{F}_{\mathsf{BB}}^{\mathsf{gv}}.\mathsf{P.Submit}$ to get back ρ. It returns (c, ρ) (note that $\rho = \perp$ when layer $= 1$).

Note that the sanitization code of lines 40, 52, and 55 of $\mathcal{F}_{\mathsf{L2Gov}}$ is made redundant by the code of line 25 of $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}$ and lines 29 and 32 of $\mathcal{F}_{\mathsf{BB}}^{\mathsf{gv}}$. Thus, the code of $\mathcal{F}_{\mathsf{L2Gov}}$ and π_{L2Gov} for the Vote interface is indistinguishable, since in all cases the encryption of the same message (x or a uniformly chosen valid message) is provided by both the simulator and protocol π_{DHE}.

$\mathcal{F}_{\mathsf{L2Gov}}.\mathsf{P.FinalizeVotes}$:

$\mathsf{Sim}(\mathsf{P.FinalizeVotes})$: The simulator follows the code of π_{L2Gov}. In particular, it reads BB_1 from the smart contract

on $\mathcal{G}_{\mathsf{MC}}$ and derives BB_2 from $\mathcal{F}_{\mathsf{BB}}^{\mathsf{gv}}$.S.FormBB. It de-duplicates BB_2 and then computes the encrypted tally CT using $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}$.S.Eval. It creates a proof of correctness of CT using $\mathcal{F}_{\mathsf{iNIZK}}^{\mathsf{gv}}[\mathbf{BBM}, \mathcal{F}_{\mathcal{R}_{\mathsf{BBM}}}]$.S.Prove and then signs it before submitting it to $\mathcal{G}_{\mathsf{MC}}$.

$V' \leftarrow \mathsf{Sim}(\mathsf{P.FinalizeVotes})$: The simulator follows the code of the π_{L2Gov} protocol and returns $\mathsf{BB}_1 \cup \mathsf{BB}_2$. Note that both BB_1 and BB_2 are de-duplicated in this code.

$\mathsf{Sim}(\mathsf{P.FinalizeVotes})$: The simulator performs all network activities here.

The sanitization code of lines 69-71 in $\mathcal{F}_{\mathsf{L2Gov}}$ is made redundant by the protocol code that sanitizes both BB_1 and BB_2 (lines 66, 69, 70 of Prot. 21). Thus, the code of $\mathcal{F}_{\mathsf{L2Gov}}$ and π_{L2Gov} for the FinalizeVotes interface is indistinguishable.

$\mathcal{F}_{\mathsf{L2Gov}}$.P.CheckInc

$b \leftarrow \mathsf{Sim}(\mathsf{P.CheckInc}, \mathsf{Q}, \psi)$: The simulator follows the code of π_{L2Gov} to parse ψ into (c, ρ), get an inclusion proof π using $\mathcal{F}_{\mathsf{BB}}^{\mathsf{gv}}$.P.GetIncProof, and obtain the verification bit from $\mathcal{F}_{\mathsf{BB}}^{\mathsf{gv}}$.P.VerifyIncProof on π. It overwrites this bit, if necessary, following the functionality code (also explained in detail below).

The sanitization code of lines 84 and 88-89 in $\mathcal{F}_{\mathsf{L2Gov}}$ is made redundant by the code in lines 78-79, 85, 95-96 of $\mathcal{F}_{\mathsf{BB}}^{\mathsf{gv}}$. Thus, the code of $\mathcal{F}_{\mathsf{L2Gov}}$ and π_{L2Gov} for the CheckInc interface is indistinguishable.

$\mathcal{F}_{\mathsf{L2Gov}}$.P.Complain()

$\mathsf{Sim}(\mathsf{P.Complain}, \mathsf{Q}, \psi)$: The simulator follows the code of π_{L2Gov} to submit a complaint transaction on $\mathcal{G}_{\mathsf{MC}}$.

The code of $\mathcal{F}_{\mathsf{L2Gov}}$ and π_{L2Gov} for the Complain interface is indistinguishable, as a correctly submitted complaint is added by the $\mathcal{G}_{\mathsf{MC}}$ smart contract (lines 97-98).

$\mathcal{F}_{\mathsf{L2Gov}}$.P.Resolve

$\mathsf{Sim}(\mathsf{P.Resolve}, \mathsf{Q}, \psi)$: The simulator follows the code of π_{L2Gov} to check the correctness of the complaint and derives a proof of inclusion using $\mathcal{F}_{\mathsf{BB}}^{\mathsf{gv}}$.P.GetIncProof, signs it, and then submits it to $\mathcal{G}_{\mathsf{MC}}$.

The code of $\mathcal{F}_{\mathsf{L2Gov}}$ and π_{L2Gov} for the Resolve interface is indistinguishable, as a correctly submitted resolution causes the complaint to be deleted by the $\mathcal{G}_{\mathsf{MC}}$ smart contract (lines 105-106).

$\mathcal{F}_{\mathsf{L2Gov}}$.P.Tally:

$\mathsf{Sim}(\mathsf{P.Tally})$: The simulator follows the code of π_{L2Gov} to read CT from $\mathcal{G}_{\mathsf{MC}}$ and compute the tally and the proof π_{tally} using $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}$.Decrypt. It signs them and then sends them to $\mathcal{G}_{\mathsf{MC}}$.

$\mathsf{tally}' \leftarrow \mathsf{Sim}(\mathsf{P.Tally}, \mathsf{tally}^*)$: The simulator returns the computed tally and π_{tally}.

Since P decrypts the same CT as everyone else (read from $\mathcal{G}_{\mathsf{MC}}$), and line 67 of functionality $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}$ ensures that F is used to evaluate the result, the sanitization of line 118 in $\mathcal{F}_{\mathsf{L2Gov}}$ has no effect on the simulator's output. Thus, the code of $\mathcal{F}_{\mathsf{L2Gov}}$ and π_{L2Gov} for the Tally interface is indistinguishable.

$\mathcal{F}_{\mathsf{L2Gov}}$.P.GetResult:

$\mathsf{res}' \leftarrow \mathsf{Sim}(\mathsf{P.GetResult})$: The simulator follows the code of π_{L2Gov} to read the result from $\mathcal{G}_{\mathsf{MC}}$.

Line 123 of $\mathcal{F}_{\mathsf{L2Gov}}$ is redundant, as the smart contract already performs that check before storing the result. Thus, the code of $\mathcal{F}_{\mathsf{L2Gov}}$ and π_{L2Gov} for the GetResult interface is indistinguishable.

We explain below how the simulator responds to the environment's queries on the globally verifiable interfaces. Note that the simulator, as part of the simulation of various hybrids, maintains internal lists that it uses to adjust the adversary's responses when required.

$\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}[\mathbf{P}', \mathbf{P}']$.P.Verify:

$b \leftarrow \mathsf{Sim}(\mathsf{P.Verify}, \mathsf{vk}, m, \sigma)$: The simulator calls the adversary on $(\mathsf{P.Verify}, \mathsf{vk}, m, \sigma)$ to receive the verification bit. If $(\mathsf{vk}, m, \sigma) \in \mathsf{Ver}_{\mathsf{int}}$, then it overwrites the adversary's bit with 1. Note that, since no keys are generated in $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{gv}}$, the list VK remains empty and the bit will not be overwritten back with a 0.

$\mathcal{F}_{\mathsf{PKI}}^{\mathsf{gv}}[\mathbf{P}']$.P.CertVerify:

$b \leftarrow \mathsf{Sim}(\mathsf{P.CertVerify}, \mathsf{pp}, \mathsf{Q}, \mathsf{vk}, \mathsf{cert})$: The simulator calls the adversary on $(\mathsf{P.CertVerify}, \mathsf{pp}, \mathsf{Q}, \mathsf{vk}, \mathsf{cert})$ to receive the verification bit. If $\mathsf{pp} = \mathsf{pp}^*$ and $(\mathsf{pp}^*, \mathsf{Q}, \mathsf{vk}, \mathsf{cert}) \in \mathsf{Ver}_{\mathsf{int}}$, then it overwrites the adversary's bit with 1, as this list maintains the list of registered certificates. Note that, since no keys are generated in $\mathcal{F}_{\mathsf{PKI}}^{\mathsf{gv}}$ (and thus no pp^* exists), the bit will not be overwritten back.

$\mathcal{F}_{\mathsf{iNIZK}}^{\mathsf{gv}}[\mathbf{BBM}, \mathcal{F}_{\mathcal{R}_{\mathsf{BBM}}}]$.P.Verify:

$w \leftarrow \mathsf{Sim}(\mathsf{P.Verify}, x, \pi)$: The simulator calls the adversary on $(\mathsf{P.Verify}, x, \pi)$ to receive the witness. If $(x, \pi) \in \mathsf{Ver}_{\mathsf{int}}$, then it looks up the witness w for the proof π generated for x from $\mathsf{Wit}_{\mathsf{int}}$ and overwrites the adversary's witness with w. With this correct witness, the functionality $\mathcal{F}_{\mathsf{iNIZK}}^{\mathsf{gv}}[\mathbf{BBM}, \mathcal{F}_{\mathcal{R}_{\mathsf{BBM}}}]$ computes the verification bit as 1, which matches the bit in $\mathsf{Ver}_{\mathsf{int}}$.

$\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}[\mathbf{P}, \mathbf{VC}, t_{\mathsf{VC}}, \mathsf{Val}, F, \Delta_1^{\mathsf{d}}, \Delta_2^{\mathsf{d}}, \Delta_3^{\mathsf{d}}]$:

$b \leftarrow \mathsf{Sim}(\mathsf{P.VerifyEnc}, \mathsf{pp}, c)$: The simulator calls the adversary on $(\mathsf{P.VerifyEnc}, \mathsf{pp}, c)$ to receive the verification bit. If $\mathsf{VerEnc}_{\mathsf{int}}[\mathsf{pp}, c] = 1$ or $\mathsf{Dec}_{\mathsf{int}}[\mathsf{pp}, c] \neq \perp$, then it overwrites the adversary's bit with 1. Now, since the functionality $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}$ lists are empty initially, the bit will not be overwritten back with a 0. For repeat queries, the same response is returned both by the simulator and the functionality.

$b \leftarrow \mathsf{Sim}(\mathsf{P.VerifyDec}, \mathsf{pp}, \mathsf{CT}, \mathsf{res}, \pi)$: The simulator calls the adversary on $(\mathsf{P.VerifyDec}, \mathsf{pp}, \mathsf{CT}, \mathsf{res}, \pi)$ to receive the verification bit. If $\mathsf{Ver}_{\mathsf{int}}[\mathsf{pp}, \mathsf{CT}, \mathsf{res}, \pi]$ is set, then the bit is overwritten by $\mathsf{Ver}_{\mathsf{int}}[\mathsf{pp}, \mathsf{CT}, \mathsf{res}, \pi]$. Otherwise, if $\mathsf{Res}_{\mathsf{int}}[\mathsf{pp}, \mathsf{CT}]$ is set and does not equal res, the bit is overwritten by 0. Now, since the functionality $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}$ lists are empty initially, the bit will not be overwritten back with a different value. For repeat queries, the same response is returned both by the simulator and the functionality.

$(\mathsf{CT}, \mathsf{M}) \leftarrow \mathsf{Sim}(\mathsf{P.Eval}, \mathsf{pp}, B)$: The simulator calls the adversary on $(\mathsf{P.Eval}, \mathsf{pp}, B)$ to receive $(\mathsf{CT}, \mathsf{M})$. For all $c \in B$ such that $\mathsf{M}_{\mathsf{int}}[c] \neq \perp$, it overwrites the

adversary-provided $\mathsf{M}[c]$. For the remaining $c \in B$, if $\mathsf{Val}(\mathsf{M}[c]) = 1$, it computes $\mathsf{res} \leftarrow F(\mathsf{M}[c] : c \in B)$. If res was computed in the previous step, it updates its list $\mathsf{Res}_{\mathsf{int}}[\mathsf{pp}, \mathsf{CT}] \leftarrow \mathsf{res}$. It returns $(\mathsf{CT}, \mathsf{M})$. Now, the functionality $\mathcal{F}_{\mathsf{DHE}}^{\mathsf{gv}}$ lists are empty initially, so it accepts the same CT as the simulator and computes the same result res. For repeat queries, since the simulator maintains consistency across lists as does the functionality, the functionality does not overwrite the simulator's output.

$\mathcal{F}_{\mathsf{BB}}^{\mathsf{gv}}[\mathbf{P}, \mathbf{BBM}, t_{\mathsf{BB}}, \mathsf{Val}_{\mathsf{BB}}^{1}, \Delta, \Delta_1^{\mathsf{b}}, \Delta_2^{\mathsf{b}}, \Delta_3^{\mathsf{b}}, \Delta_4^{\mathsf{b}}]$:

$\underline{b \leftarrow \mathsf{Sim}(\mathsf{P.VerifyRec}, (\mathsf{Q}, c), \rho)}$: The simulator calls the adversary on $(\mathsf{P.VerifyRec}, (\mathsf{Q}, c), \rho)$ to receive the verification bit. If $\mathsf{VerRec}_{\mathsf{int}}[(\mathsf{Q}, c), \rho]$ is set, then the bit is overwritten by $\mathsf{VerRec}_{\mathsf{int}}[(\mathsf{Q}, c), \rho]$. Otherwise, if $|\mathbf{BBM} \cap \mathbf{C}| \leq t_{\mathsf{BB}}$, the bit is overwritten by 0. Now, since the functionality $\mathcal{F}_{\mathsf{BB}}^{\mathsf{gv}}$ lists are empty initially, the bit will not be overwritten back with a different value. For repeat queries, the same response is returned both by the simulator and the functionality.

$\underline{b \leftarrow \mathsf{Sim}(\mathsf{P.VerifyIncProof}, \mu, (\mathsf{Q}, c), \pi)}$: The simulator calls the adversary on $(\mathsf{P.VerifyIncProof}, \mu, (\mathsf{Q}, c), \pi)$ to receive the verification bit. If $\mathsf{VerifyIncProof}_{\mathsf{int}}[\mu, (\mathsf{Q}, c), \pi]$ is set, then the bit is overwritten by $\mathsf{VerifyIncProof}_{\mathsf{int}}[\mu, (\mathsf{Q}, c), \pi]$. Otherwise, if $\mathsf{Set}_{\mathsf{int}}[\mu]$ is set and $(\mathsf{Q}, c) \notin \mathsf{Set}_{\mathsf{int}}[\mu]$, the bit is overwritten by 0. Now, since the functionality $\mathcal{F}_{\mathsf{BB}}^{\mathsf{gv}}$ lists are empty initially, the bit will not be overwritten back with a different value. For repeat queries, the same response is returned both by the simulator and the functionality.

$\underline{\mu \leftarrow \mathsf{Sim}(\mathsf{P.Com}, \mathsf{BB})}$: The simulator calls the adversary on $(\mathsf{P.Com}, \mathsf{BB})$ to receive μ. If $\mu_{\mathsf{int}}[\mathsf{BB}]$ is set, then μ is overwritten by $\mu_{\mathsf{int}}[\mathsf{BB}]$; else it updates its $\mathsf{Set}_{\mathsf{int}}$ and μ_{int} lists appropriately. Now, since the functionality $\mathcal{F}_{\mathsf{BB}}^{\mathsf{gv}}$ lists are empty initially, μ will not be overwritten back with a different value. For repeat queries, the same response is returned both by the simulator and the functionality.

This completes the description of the simulator Sim. Since the simulator completely follows the code of the protocol π_{L2Gov}, additionally maintaining internal lists for consistency across all interfaces, the indistinguishability of $(\pi_{\mathsf{L2Gov}}, \mathcal{A})$ and $(\mathcal{F}_{\mathsf{L2Gov}}, \mathsf{Sim})$ follows.

□