Privacy Preserving Opinion Aggregation

Aggelos Kiayias [1,2], Vanessa Teague [3,4], and Orfeas Stefanos Thyfronitis Litos [5,6]

Abstract: There are numerous settings in which people's preferences are aggregated outside of formal elections, and where privacy and verification are important but the stringent coercion-resistant properties of government elections do not apply, a prime example being social media platforms. In contrast to the centrally organised, single-shot elections on which most of the literature is focused, our settings of interest often have no trusted authority and are iterative, i.e., publish partial tallies before all votes are received.

In this work, we explore opinion aggregation in a decentralised, iterative setting. In our novel protocol, randomly-chosen parties take turns to decrypt aggregated votes in an incentive-driven manner. Our simple construction leverages homomorphic encryption and secret sharing in a novel manner to provide security of the tally, public and individual verifiability, as well as vote privacy and liveness guarantees. Contrary to existing voting schemes, our protocol is ideal for the iterative setting.

Keywords: e-voting, content curation, privacy, decentralisation

1 Introduction

Motivated by the need for decentralised crowdsourced content curation, in this work we construct and analyse a protocol that enables a group of non-hierarchical parties to aggregate their opinions on a single predefined topic in a privacy-preserving and timely manner. Each party may vote once, choosing among upvote (+1), downvote (−1), or abstain (0). Unlike existing e-voting protocols, our construction is intentionally not *single-shot*, i.e., it does not keep results hidden until after all votes are cast, but instead reveals them dynamically.

Our scheme is thus suitable for aggregating user upvotes and downvotes on social media content on decentralised platforms such as Lens [22], Farcaster [20], Steemit [18], or even centralised ones such as Reddit [Be11] in an online fashion whilst protecting user privacy from a central server and potential data breaches. Such settings typically allow users to post, view and rate content continuously. User-generated ratings are used to sort content, therefore ratings have to be constantly updated.

[1] University of Edinburgh, akiayias@inf.ed.ac.uk, https://orcid.org/0000-0003-2451-1430
[2] IOG, akiayias@inf.ed.ac.uk, https://orcid.org/0000-0003-2451-1430
[3] University of New South Wales, vanessa@thinkingcybersecurity.com, https://orcid.org/0000-0003-2648-2565
[4] Thinking Cybersecurity Pty. Ltd., vanessa@thinkingcybersecurity.com, https://orcid.org/0000-0003-2648-2565
[5] University of Warwick, orfeas.litos@proton.me, https://orcid.org/0000-0002-8229-297X
[6] Common Prefix, orfeas.litos@proton.me, https://orcid.org/0000-0002-8229-297X

Our threat model assumes a slowly adaptive malicious adversary that can corrupt and fully control up to a fixed number of parties. We assume fully asynchronous communication: the adversary can inspect, edit, reorder, delay, or drop all messages between parties (Sec. 2.1). Our scheme guarantees (i) *Liveness*, i.e., that votes will be processed in a timely manner, (ii) the protection of *vote privacy* from malicious attacks, (iii) *individual verifiability* in the sense that each (electronic) voter can verify the inclusion of their vote in the tally, and (iv) *public verifiability*, i.e., ability for an external observer to verify the overall correctness of the tally [Co16].

There is no obvious way to adapt a usual single-shot e-voting protocol to provide intermediate tallies, since it is explicitly forbidden. Sequentially composing multiple single-shot protocols comes with its own set of challenges, such as ascertaining that the same party does not vote across protocol invocations. We therefore believe that the problem at hand is of significant individual importance to deserve a direct treatment.

High-level Construction. Each party can at any time cast a single vote on the topic at hand along with a zero-knowledge proof of validity to a suitable smart contract on a blockchain. Each set of B consecutive votes by different parties constitutes a *batch*. A randomly chosen subset of the parties called *decryptors* decrypt each batch and publish the result, which can be used by anyone to calculate the sum of the votes in the batch. All votes of a batch have the same n decryptors. Looking ahead, each vote draws its privacy by hiding among other votes in its batch. Every party can act both as a decryptor and as a voter during a protocol execution.

At the beginning of each batch, the smart contract samples n *decryptors* from the set of parties in a verifiably random fashion — decryptors are fixed for the batch. Voting is done as follows: The voter secret-shares its vote using t-out-of-n Shamir's Secret Sharing [Sh79], creating one share for each of the n decryptors. Next, the voter encrypts each share with the public key of the corresponding decryptor, using an encryption scheme that is homomorphic for at least B additions, e.g., [Pa99], or the exponential form of [El85], and publishes the encrypted shares to the smart contract.

Each honest party also periodically checks whether it is the decryptor of a batch. In this case, it adds together its B ciphertexts, each coming from a different voter of the batch, and decrypts and publishes the resulting aggregate share on the smart contract.

Anyone can use t of the (up to) n aggregate shares to reconstruct the sum of said B votes. This approach ensures that only aggregate votes can be decrypted while individual votes remain private. A block diagram of the basic operations for parameters $B = t = n = 2$ is shown in Fig. 1. The parameters t, n, and B can be tuned to achieve different tradeoffs, see Sec. 6. Privacy relies on n as the security parameter, i.e., n needs to be large.

Core Homomorphic Technique. Why does reconstructing from sums of shares work? The answer lies in the additive-homomorphic property of encryption and secret sharing,

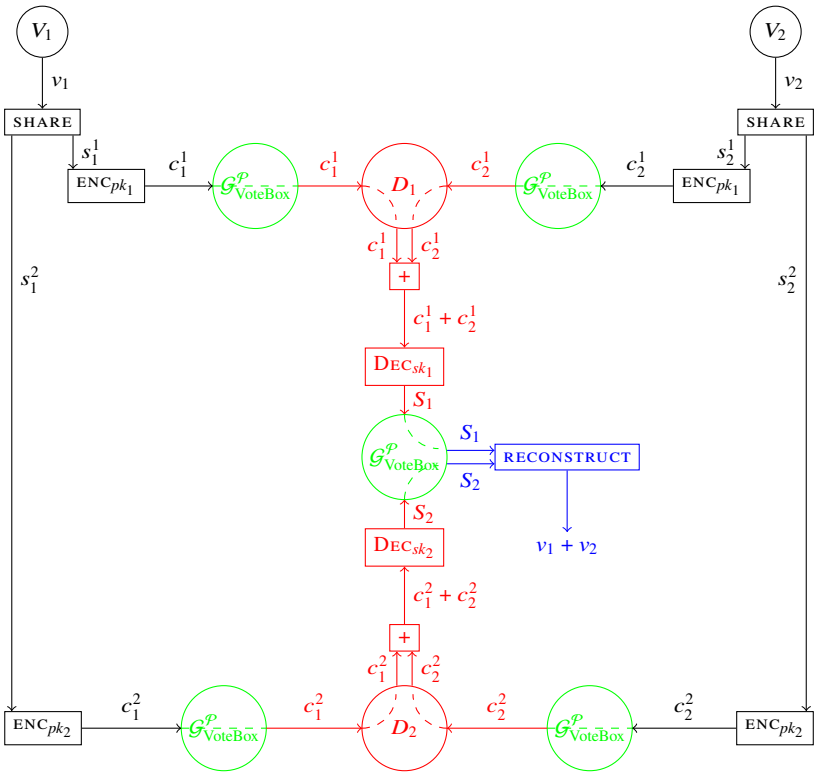

Fig. 1: $B = 2$ voters (V_1, V_2), $n = 2$ decryptors (D_1, D_2), 2-of-2 secret sharing. Notation in (1). Actions in black executed by voters, red by decryptors, blue by tallying parties. D_1 cannot reconstruct v_1 or v_2 without colluding with D_2.

detailed in Sec. 2.2. We here see an example with two voters V_1 and V_2 with votes v_1 and v_2 respectively, two decryptors D_1 and D_2 with keypairs (sk_1, pk_1) and (sk_2, pk_2) respectively, 2-of-2 secret sharing, and a tallying party T. T obtains the sum of the two votes without learning either one.

$$
\begin{aligned}
V_1: &\quad (s_1^1, s_1^2) \leftarrow \textsc{share}(v_1) \\
&\quad c_1^1 \leftarrow \textsc{enc}_{pk_1}(s_1^1), c_1^2 \leftarrow \textsc{enc}_{pk_2}(s_1^2) \\
V_2: &\quad (s_2^1, s_2^2) \leftarrow \textsc{share}(v_2) \\
&\quad c_2^1 \leftarrow \textsc{enc}_{pk_1}(s_2^1), c_2^2 \leftarrow \textsc{enc}_{pk_2}(s_2^2) \\
D_1: &\quad S_1 \leftarrow \textsc{dec}_{sk_1}(c_1^1 + c_2^1) \overset{\text{hom. enc.}}{=} s_1^1 + s_2^1 \\
D_2: &\quad S_2 \leftarrow \textsc{dec}_{sk_2}(c_1^2 + c_2^2) \overset{\text{hom. enc.}}{=} s_1^2 + s_2^2 \\
T: &\quad \textsc{reconstruct}(S_1, S_2) \overset{\text{hom. SS}}{=} v_1 + v_2
\end{aligned}
\tag{1}
$$

This approach achieves voter privacy, as individual votes are not published and no set of less than t decryptors have enough shares by any one voter to reconstruct its vote. It is enough for t of the n decryptors to be honest. Its main novelty is that it does not require coordination (e.g., distributed key generation) among decryptors similarly to [Ga24] for threshold signatures, while maintaining its threshold nature (thus withstanding corruptions). The system parameters are chosen so that it is exceedingly unlikely for any single vote of the entire execution to be secret-shared to t or more malicious players.

Related Work. The field of content curation studies how opinions can be combined to produce an aggregated outcome, generally with expressive preferences, ongoing interaction, and little focus on privacy or the possibility of manipulation [AHM14; Cr20; Da10; Ki19; SSG12; Yu09]. Content curation consists of algorithms and protocols that ensure users of a system, commonly a social media platform, access the most relevant and useful content. The constant inflow of content in such platforms precludes manual classification and ranking, and crowdsourcing the procedure by taking into account users' judgement can vastly improve the quality of curation. Therefore our voting protocol, which is suitable for privately aggregating the opinions of parties in an online fashion (as opposed to single-shot voting protocols), stands at the intersection of voting and content curation.

Internet voting schemes for public elections typically have a designated set of election administrators who are, at least, trusted for privacy [CGS97; Di03; FOO92]. These schemes scale to a very large number of voters and a modest number of authorities, often with some redundancy in case some of them fail or are corrupted. In this work, no such distinguished set of parties exists, strengthening our privacy guarantees. Among voting schemes with publicly verifiable tallies, some focus on strong coercion resistance properties [JCJ05], which our work does not offer. Other schemes focus on allowing the voter to verify that their vote is cast as they intended from an untrusted device [Ad08; CGT18; Ch16; Ha20; KSW05; KZZ17; RRI16]. In this work, we do not distinguish the intentions of a human voter from the intentions of their device. We therefore do not consider schemes based on tracker codes [JRF09; RRI16] which are mainly focused on easy verification by human users, generally in return for requiring some non-collusion assumptions among the authorities.

We also do not use the techniques for cast-as-intended verification designed for pollsite e-voting schemes [Be13; CRS05; Cu15; KZZ15]

Boardroom voting schemes [Gr04; Ku14; KY02] are designed for a relatively small group of voters to act as equals: all voters participate in both voting and tallying. Such schemes can be publicly verifiable, but suffer from robustness issues and may fail if any party refuses to decrypt. They are thus unsuitable for large groups.

Previous voting schemes have also combined homomorphic encryption and secret sharing but in different ways. In [BT94] each voter has to run a separate interactive proof protocol with each tallying authority. In [BY86; Li19] a single malicious teller/committee member can cause an election to fail. In [CGS97; ZOB19], which use robust threshold encryption instead of secret sharing, the authorities need to run an interactive distributed key generation protocol [Pe91]. The protocol of [Sc99] is similar to ours, but relies on talliers running the reconstruction protocol. Thanks to our specific technique, our protocol has none of these limitations, which would be detrimental in our context.

Zhang et al. [ZOB19] show how to use a distributed protocol to select a verifiably-random subset (weighted by *stake*) to act as a decryption committee. Voting occurs in epochs, after which a new decryption committee is chosen. Their scheme achieves slightly different goals from ours, as it incorporates a vote weighting mechanism based on each voter's stake, along with a stake delegation mechanism. Furthermore it does not provide a method that overcomes the *single-shot* limitation when revealing the aggregate votes on any particular topic; it instead provides a governance system with sequential epochs, where each topic is proposed and voted for in a single epoch. A single epoch may handle multiple independent topics.

Our setting and contribution. Our voting protocol (informally presented in Sec. 4 and formally defined in $\Pi_{\text{vote}}^{B,n,t}$, Fig. 13, Sec. 9) is run by a fixed set of parties, a subset of which are randomly chosen to act as authorities. Every party can vote once whenever they wish, but the authorities' roles are allocated to a constantly refreshed subset of parties. Parties can be both voters and authorities. Decryption happens whenever the not-yet-tallied votes form a large enough anonymity set, therefore providing *privacy* (Thm. 1). In this respect, we improve upon boardroom voting in that the need for separate authorities is avoided while robustness against misbehaving parties is guaranteed. Furthermore, our scheme provides *individual* and *public verifiability*. Looking ahead, both are achieved thanks to the public nature of the smart contract (Sec. 5.1). We also achieve *liveness* (Thm. 2), informally meaning that (under reasonable assumptions) every vote will be promptly included in the total tally. We further define opinion aggregation for a single topic as an ideal functionality $\mathcal{F}_{\text{vote}}^{B,n,t}$ (Fig. 12, Sec. 8) which is provably (Thm. 3) realised by $\Pi_{\text{vote}}^{B,n,t}$.

We achieve this via a novel combination of homomorphic encryption with secret sharing that does not require interaction among decryptors while withstanding $n - t$ corruptions.

Unfortunately our protocol is not *receipt-free*, i.e., a voter can be coerced to prove how they voted. Furthermore honest parties need to interact with the system after they vote, which however we deem an acceptable tradeoff in the era of always-on mobile applications. In order to motivate parties to stay online, we also propose a suitable incentive mechanism. Lastly, we provide an implementation of the smart contract that stores aggregate vote shares (Appx. D).

Fairness [FOO92; KY02], i.e., the guarantee that prior cast votes do not influence future votes, is not sought after on purpose. Indeed, in the setting of social media it is beneficial to have an up-to-date view of the public's reaction to a specific topic. This can offer a current and realistic pulse of the public's opinion, help users decide which posts to pay attention to, and ultimately help them choose how to vote without enabling more vote manipulation than existing platforms.

We also define *continuous* opinion aggregation as a protocol in which a stream of topics is presented and parties can vote for each. One independent execution of the protocol per topic trivially realises it. Our work paves the way for decentralised content curation on realistic social media platforms.

2 Preliminaries

2.1 Execution, Network, and Threat Model

We use the execution model of Universal Composition [Ca01], exploiting its clearly defined and widely used entities (*Interactive Turing Machines* – ITMs) and interactions between them. Our security treatment however follows the standalone simulation paradigm [Li17], leaving a UC analysis as future work (Sec. 10). The environment $\mathcal{E}$ is tasked with choosing voters' votes. Both $\mathcal{E}$ and the adversary $\mathcal{A}$ internally execute arbitrary *probabilistic polynomial-time* code. We define the *real-world protocol* $\Pi_{\text{vote}}^{B,n,t}$ in Sec. 9 and the *ideal-world functionality* $\mathcal{F}_{\text{vote}}^{B,n,t}$ in Sec. 8 so that, from the point of view of $\mathcal{E}$ and $\mathcal{A}$, these two are *computationally indistinguishable* (Thm. 3), i.e., $\mathcal{E}$ and $\mathcal{A}$ cannot ascertain whether they are in the ideal world, interacting with a single ITM running $\mathcal{F}_{\text{vote}}^{B,n,t}$, or in the real world, interacting with a set of ITMs each running $\Pi_{\text{vote}}^{B,n,t}$. Complementing simulation-based security, we also use property-based security for the most salient properties: *privacy* (Thm. 1) and *liveness* (Thm. 2).

Network communication is fully asynchronous. In other words, the *adversary* $\mathcal{A}$ fully controls the network: $\mathcal{E}$ and $\mathcal{A}$ can freely send messages to any ITM, including to each other, $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ and $\mathcal{G}_{\text{PKI}}$, but not to $\mathcal{F}_{\text{proof}}$ or $\mathcal{F}_{\text{CRS}}$ (defined in Sec. 2.2). The latter are *local* functionalities and thus can only be accessed by protocol parties. Parties' messages are mediated by $\mathcal{A}$, who can inspect, edit, reorder, or delay messages before delivering them, or even completely drop them.

As for the threat model, it is as follows: Initially all parties are honest. As we saw earlier, B is the batch size. Every time B votes are submitted, $\mathcal{A}$ can designate one or more honest parties for corruption. These parties come under the full control of $\mathcal{A}$ for the remainder of the execution right after the next batch of decryptors is chosen (Sec. 4). $\mathcal{A}$ can corrupt at most s parties in total throughout the entire execution, where s is a system parameter. This is the slowly adaptive corruption setting [PS17]. Looking ahead, s is relevant exclusively for deciding parameters t and n of the secret sharing scheme given the number of parties $|\mathcal{P}|$ (cf. Thm. 1). Honest parties remain online and active in their roles as voters and decryptors throughout the execution.

2.2 Building Blocks

Our protocol builds upon two homomorphic cryptographic primitives: firstly, it uses an additive-homomorphic public key encryption scheme with algorithms $\langle \textsc{keyGen}, \textsc{enc}, \textsc{dec}, \textsc{pk} \rangle$, realisable by, e.g., Paillier's [Pa99] or the exponential version of ElGamal's encryption schemes [El85]. The homomorphic property of encryption is $\textsc{dec}_{sk}(\textsc{enc}_{pk}(m_1) + \textsc{enc}_{pk}(m_2)) = m_1 + m_2$.

Secondly, it uses a t-of-n additive-homomorphic secret sharing scheme with algorithms $\langle \textsc{share}, \textsc{reconstruct} \rangle$, realisable by, e.g., Shamir's scheme [Sc99; Sh79]. $\textsc{share}(m)$ returns *n shares* $s_1, \ldots, s_n$ of the secret m, typically sent to different parties. When input any t of those, $\textsc{reconstruct}$ returns m. The homomorphic property of secret sharing in case $t = n = 2$ is $(s_1^1, s_1^2) \leftarrow \textsc{share}(m_1), (s_2^1, s_2^2) \leftarrow \textsc{share}(m_2) \Rightarrow \textsc{reconstruct}(s_1^1 + s_2^1, s_1^2 + s_2^2) = m_1 + m_2$.

We also leverage a number of cryptographic constructions, modelled as ideal functionalities. We define a public key infrastructure (PKI) that stores parties' keypairs in $\mathcal{G}_{\text{PKI}}$, Fig. 15. It is responsible for generating keys and only shows the private key to its owner, whereas it may send the public key of any player to any other player. This modelling simply aids the separation of concerns and does not impose a strong assumption, as the realisation of $\mathcal{G}_{\text{PKI}}$ is as simple as having each party generate its keypair locally and publish its public key to a suitable smart contract running on a Turing-complete, decentralised blockchain, such as Ethereum [Wo14] or Cardano [Ch19] (Appx. D). This ensures no central authority can influence the result, i.e., that $\mathcal{G}_{\text{PKI}}$ is decentralised.

Furthermore, this work uses an idealised non-interactive zero-knowledge proof system for proving vote and share validity, defined in $\mathcal{F}_{\text{proof}}$, Fig. 10. It ensures that every vote has a value among -1, 0, 1, and that it has been secret-shared and encrypted correctly without leaking any private information. $\mathcal{F}_{\text{proof}}$ additionally ensures that every aggregate share submitted by a decryptor is the result of the correct decryption of a correct sum of valid ciphertexts. $\mathcal{F}_{\text{proof}}$ can be instantiated with, e.g., one of Groth's schemes [Gr16] — we point out that [Gr16] needs an trusted setup, which can be done once per deployment (or even

once for multiple deployments). We recommend including as many protocol participants as possible in the setup to reduce trust requirements.

Relatedly, we use a *common reference string* (CRS) which is a public key of the aforementioned encryption scheme, abstracted via $\mathcal{F}_{\text{CRS}}$, Fig. 9. An entity that knows the corresponding secret key can prove arbitrary statements to $\mathcal{F}_{\text{proof}}$. $\mathcal{F}_{\text{CRS}}$ is realisable via a *multiparty computation* protocol [Da12] that runs KEYGEN, sends the public key to all parties and discards the secret key.

3 Continuous Opinion Aggregation

Definition 1 *A scheme for* single-topic continuous opinion aggregation *accepts a single topic, a set of voting parties, and up to one up/down/abstain vote per party. It periodically publishes a tally of the votes.*

The detailed functional behaviour is specified in $\mathcal{F}_{\text{vote}}^{B,n,t}$ (Fig. 12). Honest parties should vote "abstain" when they are indifferent to avoid leaking which topics they have an opinion on and to aid in quicker batch completion.

In practice, social media platforms display to users an endless stream of posts, i.e., topics. Definition 2 provides a suitable scheme:

Definition 2 *A scheme for* multi-topic continuous opinion aggregation *(MTCOA) accepts a stream of topics, a stream of parties per topic, and up to one up/down/abstain vote per party per topic. It periodically publishes a vote tally for each topic.*

MTCOA can be instantiated by simply running one independent instance of $\mathcal{F}_{\text{vote}}^{B,n,t}$ for each user-proposed topic. Due to the independence of the executions all guarantees offered by $\mathcal{F}_{\text{vote}}^{B,n,t}$ can be extended to the continuous opinion aggregation scheme. A possible improvement is discussed in Sec. 10.

4 Our Construction

Let $\mathcal{P}$ be a set of parties. We assume that up to s of which may be corrupted. $\Pi_{\text{vote}}^{B,n,t}$ is parametrised by the following constants: the size B of each batch, the number n of decryptors each vote is secret-shared to, and the number t of shares needed to reconstruct a secret.

4.1 $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$

Before describing our protocol, we first introduce $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$. This is an idealised functionality to which parties publish votes and aggregate shares. $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ uses $\mathcal{F}_{\text{proof}}$ to verify that

received votes and aggregate shares are valid, ensuring parties only contribute honestly to the protocol. Like $\mathcal{G}_{\text{PKI}}$, $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ can be realised by a smart contract (Appx. D) in a Turing-complete blockchain, e.g., Ethereum [Wo14] or Cardano [Ch19]. This ensures that $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ is decentralised and that the proofs it verifies are publicly verifiable. If realised via MPC, then verifiable MPC [Ba20] is needed to achieve public verifiability that extends beyond the parties actively implicated in the protocol. Both realisations ensure that $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ is decentralised.

The main task of $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ is to validate and store votes and decrypted batch shares. At the start of each batch $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ samples uniformly from all parties n decryptors. $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ ignores parties that try to vote multiple times. To vote, a party asks $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ for the current set of decryptors. Within a fixed timeframe from reading, the voter then sends its vote as encrypted shares to $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$, which verifies their correctness and stores them — if the voter fails to send its vote in time, $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ writes them off as malicious and assigns this voter's decryptors to the next voter that asks for decryptors. Once B votes with the same decryptors are collected, $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ samples a new decryptor set. This method of selecting decryptors ensures that valid batches will be decrypted in a timely manner, while a malicious party cannot sabotage the liveness of the decryption of a valid batch of votes by asking for a set of decryptors and then stalling.

A decryptor that has handled a batch (discussed later) submits the result to $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$. Once again, $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ verifies that the decrypted batch share was generated correctly and stores it. Lastly, parties may query both the encrypted vote shares and the decrypted batch shares from $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$.

By leveraging the zero-knowledge proof functionality $\mathcal{F}_{\text{proof}}$ to verify the correctness of votes and batch shares, $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ checks for and guards against all possible malicious actions, while restricting leakage to exactly the information proven by $\mathcal{F}_{\text{proof}}$, thus protecting the privacy of individual votes. This functionality has to be realised in a way that guarantees its randomness cannot be biased (e.g., by a smart contract with good quality randomness) to ensure decentralisation.

4.2 Protocol Description

We now focus on the real-world execution. Honest parties follow $\Pi_{\text{vote}}^{B,n,t}$ (Fig. 13) to fulfill both their voter and decryptor roles. In the *initialisation phase* of the execution, each party generates its keypair. This happens when $\mathcal{E}$ sends (INIT) to a party (Fig. 13, l. 1), which in turn asks $\mathcal{G}_{\text{PKI}}$ to generate and store the keypair — if $\mathcal{G}_{\text{PKI}}$ is instantiated as a smart contract (Appx. D), this costs 1 write to the latter (Fig. 15, l. 1).

Note that, even though it is possible for $\mathcal{E}$ to skip to the next phase before initialisation is complete by sending a (READ), (VOTE, v) or (DRAINBATCH) message to a party, this will lead to the first activation of $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$, (Fig. 13, ll. 6, 18 and 31 respectively), which

triggers the initialisation sequence of $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ (Fig. 11, ll. 1-9) which halts if there is any non-initialised party, leading to the voting protocol halting and all security properties being satisfied in vacuum. We therefore focus only on the cases in which initialisation of all parties takes place before moving on to the next phase.

Execution then moves to the *voting phase*. $\mathcal{E}$ may at any time send an instruction to any protocol party: (i) (VOTE, v) instructs it to act as a voter, i.e., cast vote v, (ii) (DRAINBATCH) tells it to act as a decryptor, i.e., attempt decryption of a full batch of votes, and (iii) (READ) has it act as a tallying party, i.e., read batch shares and reconstruct decrypted vote tallies. These are described in detail below. Note that $\mathcal{E}$ decides the vote of honest parties, since $\mathcal{E}$ models, among others, the human users of a practical software implementation of our protocol.

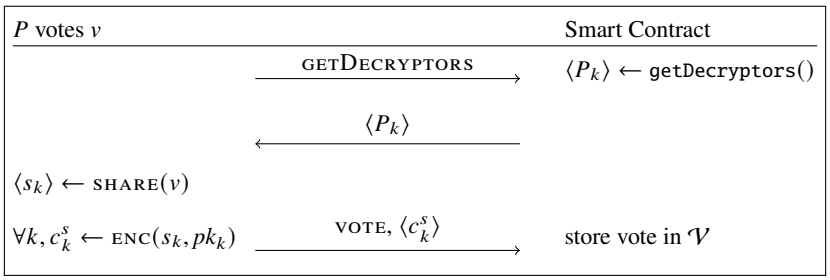

Fig. 2: Voting subprotocol (2 writes to Smart Contract). $k \in [n]$. ZK proofs elided in subprotocol figures for simplicity.

Voting subprotocol. When an honest party is instructed to vote (Fig. 2), it does so in four consecutive steps. In step 1, the party asks $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ to provide it with a set of decryptors (Fig. 13, l. 18). $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ first ensures that the party has not attempted to vote, successfully or not, in the past (Fig. 11, l. 11) and then chooses the decryptor set according to the following logic: If the previous party that attempted to vote failed to do so (only possible for a malicious party, see below), $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ reuses the same set of decryptors (Fig. 11, l. 17 was executed but l. 20 was not, so Fig. 11, l. 14 is not run and thus Fig. 11, l. 18 is executed with the previously stored `decryptors`). Otherwise, if a number of votes that is a multiple of B has been successfully cast up to that moment, a uniformly random n-sized subset of $\mathcal{P}$ is chosen (Fig. 8, ll. 5-6). In case the number of successfully cast votes is not a multiple of B, the previously used set of decryptors is used again (Fig. 8, l. 7). This method of selecting decryptors ensures that valid batches will be decrypted in a timely manner, while a malicious party cannot sabotage the liveness of the decryption of a valid batch of votes by asking for a set of decryptors and then stalling. More details can be found in Sec 4.1.

In step 2, the party secret-shares its vote and encrypts one share with the public key of each decryptor (Fig. 13, ll. 19-27). The randomness for the SHARE and ENC algorithms is sampled explicitly so that it can be subsequently passed to $\mathcal{F}_{\text{proof}}$. We do not show the $\mathcal{F}_{\text{proof}}$-related logic in Figs. 2–4 for simplicity.

In step 3, the party submits its plaintext vote, the ciphertexts it just built, the corresponding decryptors, the randomness it used and the CRS to $\mathcal{F}_{\text{proof}}$ (Fig. 13, l. 28), which in turn ensures that the vote is -1, 0, or 1 and that the ciphertexts have been generated correctly. $\mathcal{F}_{\text{proof}}$ then stores this fact along with the passed CRS and returns control to the party (Fig. 10, ll. 1-19). We note that in a realistic software implementation of the protocol, the party would locally generate a non-interactive zero-knowledge proof (NI-ZKP) of these facts instead, never sending its vote in plaintext form anywhere.

In step 4, the party sends the encrypted shares to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ (Fig. 13, l. 29), which then asks $\mathcal{F}_{\text{proof}}$ if the ciphertexts were generated correctly (Fig. 11, l. 19). We note that in a practical implementation, the party would instead have to pass the NI-ZKP to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ for verification instead. If verification succeeds, $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ records the vote locally (Fig. 11, l. 21). The party has now voted successfully at the cost of 2 writes to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$.

We note that in the UC execution model, a running instance T of an ITM cannot be interrupted by another machine; it is always allowed to complete its current chunk of computation. This is realistic, as a computation done by secure software is not interrupted by network messages. If T sends a message m to another machine T', then T' starts to run the code that corresponds to m. Furthermore, fail-stop failures (a.k.a. crashes) are not modelled separately, but as part of adversarial corruption.

Observe now that the communication flow when an honest party P votes is $P \to \mathcal{G}^{\mathcal{P}}_{\text{VoteBox}} \to P \to (\mathcal{G}_{\text{PKI}} \to P \to)^n \; \mathcal{F}_{\text{CRS}} \to P \to \mathcal{F}_{\text{proof}} \to P \to \mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$. There is no instant in which this flow is interrupted by $\mathcal{A}$ or $\mathcal{E}$ as the messages do not use the (adversary-controlled) network and all functionalities are always honest, therefore this flow can only be broken if the party is corrupted. This means that if a party has not successfully completed voting before a new vote arrives, then this party is corrupted. $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ can use this fact to mark a party that got the current decryptors but did not submit a vote as corrupt. In a practical implementation, this can be achieved with a reasonable timeout.

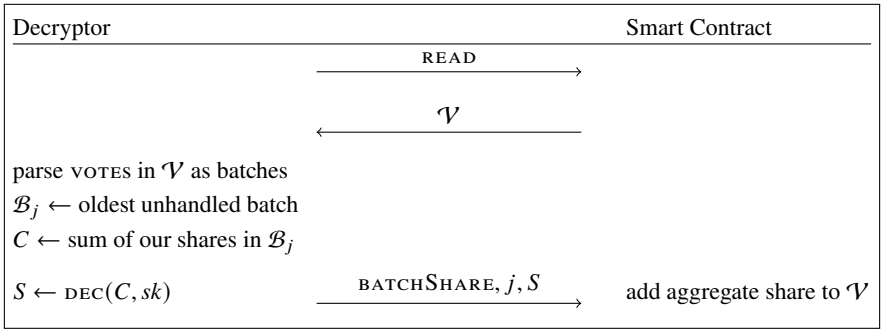

Fig. 3: Simplified batch drain subprotocol (1 write to Smart Contract)

Drain batch subprotocol. When an honest party receives (DRAINBATCH) (Fig. 3), it takes on its decryptor role. It first reads all votes from $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ (Fig. 13, l. 31). It then partitions the list of votes into batches (Fig. 13, l. 32) using the deterministic Batch algorithm (Fig. 14), which splits the list of votes into sequential batches each of size B.

The party then looks for the oldest batch for which it is a decryptor and that it has not drained yet. If it finds one, it drains it as follows (Fig. 13, l. 33): First the decryptor calculates the sum of all the ciphertexts in the batch that were encrypted with its public key (Fig. 13, l. 36). As discussed previously, these ciphertexts are guaranteed to be valid by $\mathcal{F}_{\text{proof}}$. Then the party decrypts the sum, proves to $\mathcal{F}_{\text{proof}}$ that the process was done correctly (Fig. 13, l. 39) and sends the resulting aggregate share to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ (Fig. 13, l. 40), which in turn verifies with $\mathcal{F}_{\text{proof}}$ its correctness (Fig. 11, l. 25) and stores it (Fig. 11, l. 26). Batching costs 1 write to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$.

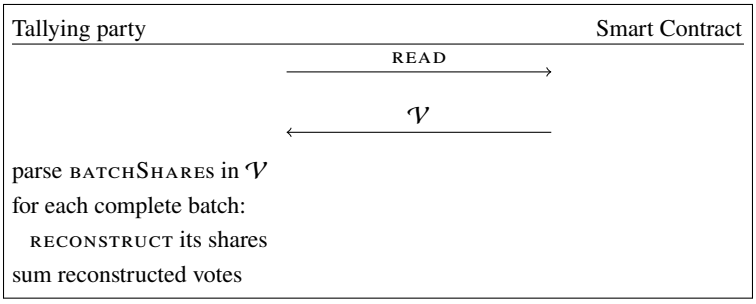

Fig. 4: Simplified tally votes subprotocol (no writes to Smart Contract)

Tallying subprotocol. Lastly, a party may receive (READ) from $\mathcal{E}$ (Fig. 4), in which case it takes on the role of a tallying party. It first reads all batch shares from $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ (Fig. 13, l. 6). For each batch, the party then reconstructs t aggregate shares from this batch to extract a valid tally. Due to the additive-homomorphic property of the encryption and secret sharing schemes, reconstructing the aggregate shares produces the sum of votes in the batch (Fig. 13, ll. 7-15). Tallying does not incur any write to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$.

Remarks. We note that $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ can be readily implemented in a blockchain that supports Turing-complete, stateful smart contracts, such as Ethereum [Wo14] or Cardano [Ch19] (a Solidity implementation is provided in Appx. D). It is an open question whether $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ can be implemented by a blockchain with a weaker scripting system such as Bitcoin [Na08]. Initialisation costs 1 on-chain transaction, voting 2, batching 1, and tallying 0. Special care should be given to ensuring that the randomness is unpredictable, in order to avoid any single party affecting the choice of decryptors. To that end, a suitable distributed protocol for implementing a randomness beacon such as [HYL20; MKS17] should be employed. Alternatively, parties can input randomness at initialisation, the sum of which can be used as a seed to a pseudorandom generator — a single honest party suffices to ensure a uniformly random seed. Since our protocol has no synchrony requirements, we do not need any

additional assumptions on top of the ones needed by the blockchain. No privacy is assumed for data in $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ or in the corresponding smart contract. To make the analysis simpler, we assume our functionalities are executed in an idealised manner and do not explicitly interact with a blockchain.

The parameter n, i.e., the number of decryptors per batch, is considered the security parameter and used as such in Thm. 1, which gives specific privacy guarantees. We remind that t is the number of honest decryptors needed per batch and s is the maximum number of total corrupted players. At a high level Thm. 1 states that, as long as $\frac{t}{n} > \frac{s}{|\mathcal{P}|}$, the probability of any one set of decryptors throughout the entire execution containing t or more corrupted parties is negligible in n. This theorem provides a useful guideline for choosing safe values for the parameters n, t given specific expectations on the number of corruptions s and number of parties $|\mathcal{P}|$. It also confirms the following appealing intuition: for the voting system to be private, the minimum required per-batch "honesty ratio" must exceed the ratio of corrupted players overall.

The batch size parameter B can be treated separately. Changing its value is connected to trading off anonymity set size for batch frequency. Indeed, for bigger values of B more votes need to be cast to complete a batch, therefore the anonymity set each party enjoys is bigger, but at the same time each party will need to wait for more votes to be submitted by others before this bigger batch is completed and the party's vote is counted. Thm. 2 proves that a party has to wait for at most $B - 1$ more votes to be cast before its vote may be counted. Sec. 6 contains further discussion on the parameters.

Properties of our construction modelled as a functionality. In the ideal world, the protocol $\Pi_{\text{vote}}^{B,n,t}$ is replaced by the functionality $\mathcal{F}_{\text{vote}}^{B,n,t}$. As we will see, we prove that $\Pi_{\text{vote}}^{B,n,t}$ realises $\mathcal{F}_{\text{vote}}^{B,n,t}$. The latter receives and handles inputs from all honest parties, whereas inputs from malicious parties are forwarded directly to the simulator $\mathcal{S}$ (Fig. 7). Likewise, an output addressed from a malicious party to $\mathcal{E}$ is passed by $\mathcal{S}$ to $\mathcal{F}_{\text{vote}}^{B,n,t}$ which subsequently forwards it to $\mathcal{E}$ unchanged. Intuitively, $\mathcal{S}$ models the ideal-world adversary, i.e., controls the corrupted parties and the network, as well as learns all non-private messages and data.

$\mathcal{F}_{\text{vote}}^{B,n,t}$ is responsible for handling the same messages from $\mathcal{E}$ that honest parties handle in the real world. In the initialisation phase, it asks $\mathcal{S}$ to generate a keypair for each party that is initialised. In the voting phase, when a party is instructed to vote, $\mathcal{F}_{\text{vote}}^{B,n,t}$ asks for and receives from $\mathcal{S}$ the corresponding vote number and decryptors, it checks that the vote number has not been reused and that the decryptor set does not include t or more malicious parties. It then stores the vote locally, before informing $\mathcal{S}$ of the successful vote. $\mathcal{S}$ submits the vote to $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$. When a party is instructed to drain a batch, $\mathcal{F}_{\text{vote}}^{B,n,t}$ finds the oldest complete, non-drained batch for this player, if any, following steps similar to the real-world protocol, and marks this batch as drained by this party. If the batch has been drained by t parties, $\mathcal{F}_{\text{vote}}^{B,n,t}$ then adds together the honest votes of the batch, asks $\mathcal{S}$ for the sum of the votes of the malicious voters of the batch, updates the locally stored results and sends the sum of the honest votes to $\mathcal{S}$. In any case, $\mathcal{F}_{\text{vote}}^{B,n,t}$ informs $\mathcal{S}$ on the outcome of the draining.

If draining succeeded, once again $\mathcal{S}$ submits the result to $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$. $\mathcal{S}$ also informs $\mathcal{F}_{\text{vote}}^{B,n,t}$ whenever a malicious party drains a batch. Lastly, when a party is instructed to read the results, $\mathcal{F}_{\text{vote}}^{B,n,t}$ simply returns them as stored.

Observe that the functionality always counts in the results all honest votes that have been included in a complete, fully drained batch. Furthermore, as we will see in the proof of Thm. 3, $\mathcal{S}$ knows the votes of all malicious parties and reports them truthfully to $\mathcal{F}_{\text{vote}}^{B,n,t}$, which in turn includes them in the results as well. These two facts together show that we have public verifiability in the ideal world, i.e. we can be certain that no votes have been neglected in the result, as long as they are part of a drained batch. Furthermore, given that Thm. 3 proves indistinguishability of the real and the ideal world without any assumptions on the number of corruptions, only having to trust the underlying ideal functionalities, we deduce that the real world protocol provides public verifiability as well. Observe that the only limitation on the number of corruptions is imposed by l. 17 of Fig. 12, which is relevant exclusively for providing the explicit privacy guarantees of Thm. 1 both in the real ($\Pi_{\text{vote}}^{B,n,t}$) and in the ideal ($\mathcal{F}_{\text{vote}}^{B,n,t}$) worlds; the indistinguishability proof would go through if this line was missing, but with any amount of corruptions allowed.

Incentivising Decryptor Participation. Our construction expects honest parties to come online frequently throughout the entire protocol execution, even after casting their vote, so that they can serve as decryptors when they are chosen by $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$. On the other hand, some users would presumably prefer to come online just to vote and be able to stay offline throughout the rest of the protocol; the aim of the protocol is to facilitate expressing user opinion, so we expect it to be used by parties that find utility in voting. A single party that stays offline except when voting would fail to fulfill its role as decryptor. This alone would not affect the protocol as long as at least t decryptors per batch behave honestly. If however all parties were to come online exclusively for voting, then no party would ever be online to act as a decryptor and therefore no batch would ever be decrypted. This situation constitutes a *tragedy of the commons*.

If the protocol is executed on a blockchain that has a cryptocurrency, then the following incentive scheme can be used to avoid this unfortunate situation: Each party pays a fee f to join. Each decryptor is paid out a set sum c as long as it publishes its decrypted sum of shares within a fixed time, e.g., 1 hour, after batch completion. The decryptor is paid irrespective of whether it is one of the first t. The values f and c are tuned so that each party that stays online throughout the execution gets 0 coins on expectation, but a party that comes online only to vote and stays offline for the rest of the execution loses coins.

More specifically, let X be the random variable that counts the number of batches for which a specific party is chosen as decryptor throughout the entire execution, let b be the total number of batches and p the probability that the party is chosen as the decryptor of a particular batch the latter stemming from Fig. 8, l. 6. Since the decryptors of each batch are chosen independently, the expected value of X is $\mathbb{E}(X) = bp$. Since we have a total of $|\mathcal{P}|$ votes and each group of B votes forms a batch, it is $b = \lfloor \frac{|\mathcal{P}|}{B} \rfloor$. If we further

assume that B divides $|\mathcal{P}|$ (as we recommend in Sec. 6), then $b = \frac{|\mathcal{P}|}{B}$. Regarding p, it is $p = \binom{|\mathcal{P}|-1}{n-1} / \binom{|\mathcal{P}|}{n} = \frac{n}{|\mathcal{P}|}$. We thus have $\mathbb{E}(X) = \frac{n}{B}$. Therefore, for any $c > 0$, if joining costs $f = c\frac{n}{B}$ coins and successfully decrypting a batch is compensated with c coins, then a party that is online throughout the execution gains on expectation 0 coins, whereas a party that comes online only to vote has to pay $c\frac{n}{B}$ coins. Realistic implementations should choose a sufficiently small c so that parties are not denied participation due to low available funds or due to fears of becoming a decryptor too few times to cover the cost of voting, but also c should be sufficiently large to ensure voting is expensive enough for most parties to prefer staying online to be compensated. This solution avoids the tragedy of the commons while leaving, on expectation, monetary value out of the equation.

A necessary sanity check is whether the contract holds enough money to compensate decryptors at every instant. Indeed, initially parties pay a total of $c|\mathcal{P}|\frac{n}{B}$ coins to join. If all n chosen decryptors decrypt correctly for all $\frac{|\mathcal{P}|}{B}$ batches, they will be compensated with a total of $c|\mathcal{P}|\frac{n}{B}$ coins. The joining fees suffice to pay all decryptors.

Note that this incentivisation scheme does not hurt the privacy of the overall protocol, as votes are still encrypted in exactly the same way and payments are independent from the content of the encrypted vote. Furthermore, the instances in which each party interacts with the system and their observability does not change. The fact that this incentivisation scheme leverages potentially traceable coins introduces a new potential privacy goal, namely to ensure funds untraceability. Extensive research [BCT21; Be14; Bo20; Bü20; Ka18; KKK21; St19; Su17; ZNP15] has been conducted on this subject and could be leveraged to this end.

5 Security, Privacy & Liveness Guarantees

Next we state the guarantees of our scheme, their proofs are in Appx. C. We first observe that the privacy of a single honest voter P may be broken if P is in a batch with $B - 1$ malicious players, as they can act honestly until learning the aggregate result of the batch and then subtract their votes from the sum. In the current protocol, $\mathcal{E}$ can arrange the vote order as it pleases, thus this attack is feasible. Still, in a practical deployment, and with a constant stream of votes, this attack would be hard to carry out. An alternative protocol could instead have $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ decide the order of players' votes, thwarting this attack against privacy. Such a measure would however impede the ability of each party to vote when it decides to (which is the case in a practical implementation of our protocol) and would cause the protocol to stall whenever the next voter happens to be unavailable. The implications of this attack depend on the probability distribution of votes, because much larger batches (anonymity sets) are required if the votes are highly biased. We therefore define privacy relative to an idealised reporting mechanism that tallies each batch.

Definition 3 (Privacy) *In a real-world execution with parties in $\mathcal{P}$ of which at most s are malicious, n decryptors per batch, and a t-of-n secret sharing scheme, let X be the event under*

which at least one of the sets of decryptors returned by $\mathsf{getDecryptors}_{n,B,\mathcal{P}}$ *contains at least* t *malicious parties. We say that* $\Pi_{\text{vote}}^{B,n,t}$ *is* private *if* $\Pr[X] \leq negl(n)$ *in* $\text{EXEC}^{\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}},\mathcal{F}_{\text{proof}}}_{\Pi_{\text{vote}}^{B,n,t},\mathcal{A},\mathcal{E}}$.

Theorem 1 (Privacy)

$\forall$ *PPT* $\mathcal{A},\mathcal{E},\forall$ *set of parties* $\mathcal{P}, \forall 0 < \alpha \leq 1 : \alpha|\mathcal{P}| \in \mathbb{N}, n = \alpha|\mathcal{P}|,$
$\forall 0 < \gamma \leq 1 : \gamma n \in \mathbb{N}, t = \gamma n, \forall 0 \leq \beta < 1 : \beta|\mathcal{P}| \in \mathbb{N}, s = \beta|\mathcal{P}|, B \in \mathbb{N},$
If $\frac{t}{n} > \frac{s}{|\mathcal{P}|}$ *then* $\Pi_{\text{vote}}^{B,n,t}$ *is private.*

Note that this does not preclude all cases in which $\mathcal{A}$ learns individual votes. For example, if all honest parties are unanimous, then $\mathcal{A}$ can deduce their votes just by looking at the aggregate tally. Differential privacy techniques, which can protect against such cases, can be orthogonally used on top of our approach.

Furthermore, if $\mathcal{A}$ has sufficient control over the network, they can stuff a batch with $B-1$ votes to learn the remaining honest vote in the batch. Since we assume $s \in O(|P|)$ corruptions, we can set B to $O(|\mathcal{P}|)$ as well (e.g., $B = \frac{|\mathcal{P}|}{100}$). This ensures that the effects of batch stuffing on privacy do not worsen as $\mathcal{P}$ increases, without affecting overall complexity. If we assume that each voter chooses when to vote independently, this change keeps the rate of tally decryptions constant in $\mathcal{P}$ (as opposed to $O(\frac{1}{\mathcal{P}})$ for constant B). In practice, batch stuffing can also be addressed by using off-the-shelf rate limiting techniques, spreading suspicious bursts of votes over many batches, or even assuming that it is statistically likely that at least two honest parties vote in rapid succession.

Theorem 2 (Liveness) *In a real-world execution, every honestly cast vote will become part of a valid decrypted batch in* $\mathcal{V}$ *of* $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ *after at most another* $B-1$ *votes for the topic at hand have been successfully cast.*

We note that no synchrony assumptions are made, therefore all party activations are controlled by $\mathcal{E}$. The latter may choose to never activate enough parties to complete a batch (with a VOTE message) or enough decryptors to decrypt it (with a DRAINBATCH message). All security properties are maintained under this strict network model. In a real software implementation these two actions would be initiated spontaneously, not at the whim of $\mathcal{E}$.

The security theorem shows that the real world is indistinguishable from the ideal one, thus the adversary cannot alter the tallies:

Theorem 3 (Security)

$$\forall \text{ PPT } \mathcal{A}, \exists \text{ PPT } \mathcal{S} : \forall \text{ PPT } \mathcal{E} \text{ it is } \text{EXEC}^{\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}},\mathcal{F}_{\text{proof}},\mathcal{G}_{\text{PKI}}}_{\Pi_{\text{vote}}^{B,n,t},\mathcal{A},\mathcal{E}} \approx \text{EXEC}^{\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}},\mathcal{G}_{\text{PKI}}}_{\mathcal{S},\mathcal{E}}.$$

5.1 Public and Individual Verifiability

We note that in no part of the security proof did we put any specific bounds on the number of corrupted players except, in the interest of privacy, to require that no set of decryptors contained t or more malicious players, a property enforced by $\mathcal{F}_{\text{vote}}^{B,n,t}$ (Fig. 12, l. 17). The indistinguishability argument would work even if this property were not enforced, allowing for a truly arbitrary number of corruptions. Furthermore, by simple inspection of $\mathcal{F}_{\text{vote}}^{B,n,t}$, we can see that honest votes in $\mathcal{F}_{\text{vote}}^{B,n,t}$ are counted correctly once in an opened batch, irrespective of the number of corrupted parties. This means that our construction enjoys public verifiability, i.e., that any observer (whether it participated in the process or not) can ascertain that the smart contract properly adds every valid vote to exactly one batch that is subsequently decrypted by sufficient decryptors.

Furthermore, individual verifiability in our scheme is very straightforward: each voter can see their vote on the smart contract and ascertain the correctness of the smart contract. (Recall that we do not distinguish a human operator from the computer that casts the vote.)

5.2 Multi-Topic Continuous Opinion Aggregation Guarantees

As discussed earlier, a multi-topic continuous opinion aggregation scheme consists of one independent $\mathcal{F}_{\text{vote}}^{B,n,t}$ execution per topic. Both the *privacy* and the *liveness* properties hold for each $\mathcal{F}_{\text{vote}}^{B,n,t}$ execution and these executions do not share any state, therefore the adversary cannot leverage information from one execution to break the guarantees of another. The properties thus carry over to each vote batch of the combined scheme. Regarding security, Thm. 3 ensures that each $\mathcal{F}_{\text{vote}}^{B,n,t}$ instance can be realised by a $\Pi_{\text{vote}}^{B,n,t}$ protocol, as long as no state is shared, e.g., parties must use independently generated keys for each $\Pi_{\text{vote}}^{B,n,t}$ execution. It can also be proven that privacy, liveness and security also hold if parties reuse public keys across $\Pi_{\text{vote}}^{B,n,t}$ executions for efficiency.

6 Recommended Parameters

We here calculate the probability of a bad event X — namely that a uniformly random set of decryptors has t or more malicious parties — for specific values of the parameters n, t, maximum total corruptions s and total players $|\mathcal{P}|$. In case X happens, $\mathcal{A}$ would be able to reconstruct votes without waiting for a batch to complete, and thus break privacy. When sampling an n-sized subset of $\mathcal{P}$, the probability of obtaining a decryptor set with at least t malicious parties is $P[X] = \frac{\sum_{i=t}^{\min(s,n)} \binom{s}{i}\binom{|\mathcal{P}|-s}{n-i}}{\binom{|\mathcal{P}|}{n}}$ and the probability of one or more bad events X occurring throughout an entire execution is upper bounded by $\lceil \frac{|\mathcal{P}|}{B} \rceil \Pr[X]$. We visualise $\lceil \frac{|\mathcal{P}|}{B} \rceil \Pr[X]$ under relevant parameter combinations when $|\mathcal{P}| = 10000$ in Fig. 5.

For example, We determine that X happens at least once throughout an execution with probability less than 0.001 for $|\mathcal{P}| \geq 8200$, $n = \lfloor \frac{|\mathcal{P}|}{100} \rfloor$, $s = \lfloor \frac{|\mathcal{P}|}{4} \rfloor$ and $t = \lfloor \frac{n}{2} \rfloor$. The provided code[7] can be used to plot and calculate exactly these probabilities for custom parameters to inform system designs.

7 Visualisation of probability of bad event

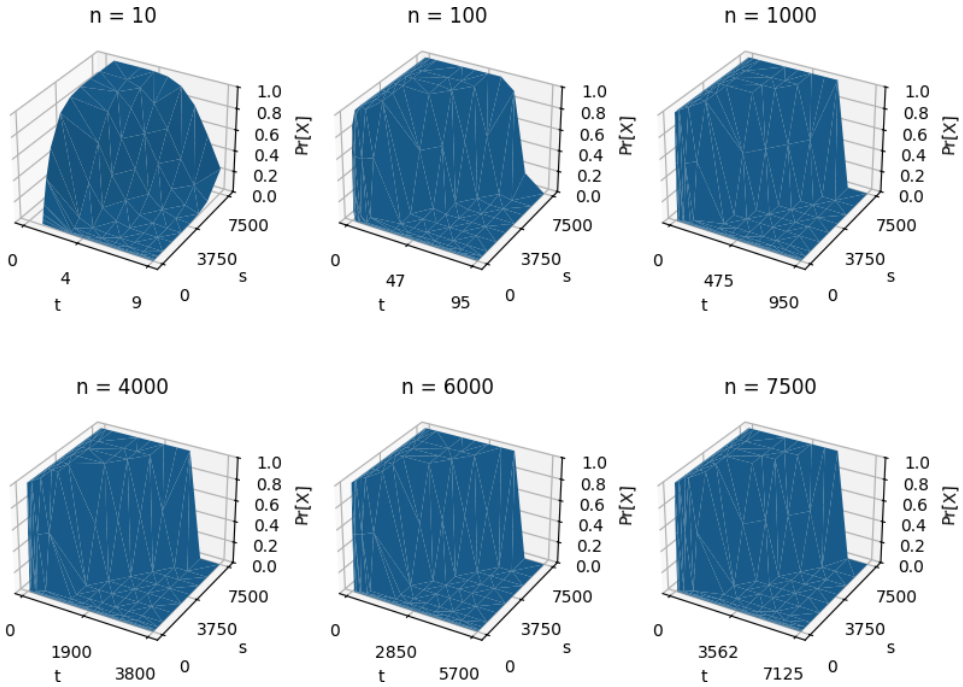

Fig. 5: Visualisation of probabilities of bad event X, i.e., choosing too many malicious decryptors (Fig. 11, l. 14) for various parameters: parties $|\mathcal{P}| = 10000$, decryptors per batch $n \in \{10, 100, 1000, 4000, 6000, 75000\}$, honest decryptors required $t \in [0, 0.95n]$, maximum overall corruptions $s \in [0, 0.75|\mathcal{P}|]$. We observe an almost step increase from Pr = 0 to Pr = 1.

For the batch size B, we recommend setting B to $O(|\mathcal{P}|)$ as well (e.g., $B = \frac{|\mathcal{P}|}{100}$). Given that assumed corruptions scale with $O(|\mathcal{P}|)$, this ensures that batch stuffing (submitting $B - 1$ adversarial votes in a batch to break the honest batch voter's privacy) does not become easier as $|\mathcal{P}|$ increases. Experiments in real-life deployments are needed to specify B as best suited to the specific application. Furthermore, if B divides $|\mathcal{P}|$, then no incomplete batch remains when all parties have voted, thus all votes are considered.

[7] https://gitlab.com/orfeasLitos/democratic-decision-making-parameters

8 Formal Description of the Ideal World

We here present the details of the local functionality $\mathcal{F}_{\text{proof}}$, the global functionality $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$, and main functionality $\mathcal{F}^{B,n,t}_{\text{vote}}$, which is realised by $\Pi^{B,n,t}_{\text{vote}}$ (detailed in Sec. 9). The supporting logic, including the global functionality $\mathcal{G}_{\text{PKI}}$ and the local functionality $\mathcal{F}_{\text{CRS}}$, can be found in Appx. A. A schematic depiction of the entities and interactions in the real and ideal worlds can be found in Figs. 6 and 7 respectively.

The t-out-of-n secret sharing scheme provides the following algorithms:

- $(s_1, \ldots, s_n) \leftarrow \textsc{share}(m; r)$ where r is sampled uniformly from $\mathcal{R}_{\text{share}}$,
- $m \leftarrow \textsc{reconstruct}(s_1, \ldots, s_t)$.

The asymmetric encryption scheme provides these algorithms:

- $(pk, sk) \xleftarrow{\$} \textsc{keyGen}()$,
- $c \leftarrow \textsc{enc}(m, pk; r)$ where r is sampled uniformly from $\mathcal{R}_{\text{enc}}$,
- $m \leftarrow \textsc{dec}(c, sk)$.

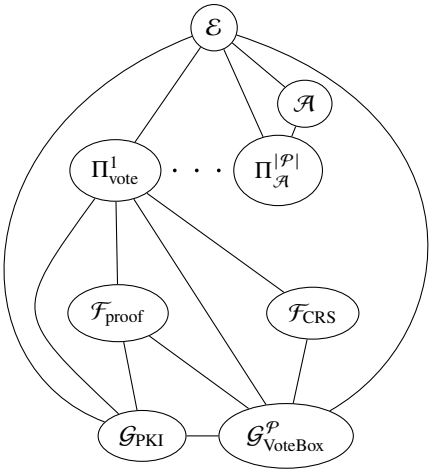

Fig. 6: Interactions in the real world. $|\mathcal{P}|$ protocol parties are involved, some of which may be corrupted (e.g., $\Pi^{|\mathcal{P}|}_{\mathcal{A}}$). The honest ones (e.g., Π^1_{vote}) run $\Pi^{B,n,t}_{\text{vote}}$. All protocol parties and the adversary $\mathcal{A}$ can interact with the 4 functionalities — here only the interactions of Π^1_{vote} are shown to avoid clutter.

$\mathsf{getDecryptors}_{n,B,\mathcal{P}}()$ is used internally by $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ to sample the decryptors of each vote.

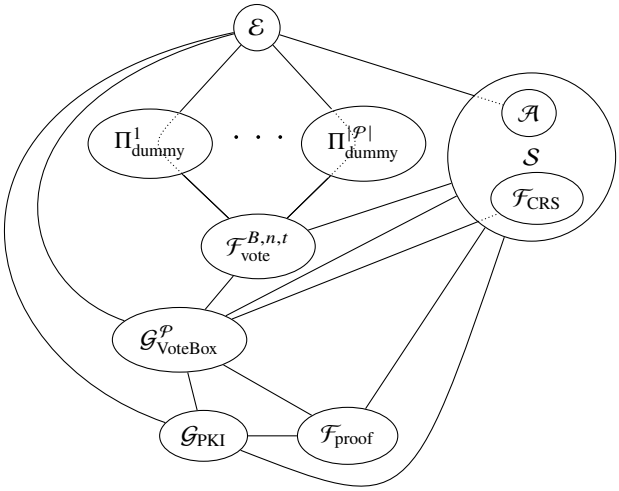

Fig. 7: Interactions in the ideal world. Real-world protocol parties are replaced with pass-through "dummy" parties. The simulator $\mathcal{S}$ subsumes the real-world adversary $\mathcal{A}$ and $\mathcal{F}_{\text{CRS}}$. $\mathcal{S}$ relays all $\mathcal{E} \leftrightarrow \mathcal{A}$ and $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}} \leftrightarrow \mathcal{F}_{\text{CRS}}$ communication.

Stateful Function $\mathsf{getDecryptors}_{n,B,\mathcal{P}}()$

1: Initialisation:
2: $r \leftarrow 0$

3: Execution:
4: $r \leftarrow r + 1$
5: **if** $r \mod B = 1$ **then**
6: $K_r \xleftarrow{\$} \binom{\mathcal{P}}{n}$ // $\binom{\mathcal{P}}{n}$ is the set of n-sized subsets of $\mathcal{P}$
7: **else**
8: $K_r \leftarrow K_{r-1}$
9: **end if**
10: **return** (r, K_r)

Fig. 8: $\mathsf{getDecryptors}_{n,B,\mathcal{P}}()$ function: samples decryptors for a vote. Used by $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$.

The $\mathcal{F}_{\text{CRS}}$ functionality initially generates a CRS public key and subsequently always returns it. The corresponding secret key is discarded. The CRS is exposed separately so that the simulator $\mathcal{S}$ can replace it with a key of its own in the proof of Thm. 3 following the simulation-extractable NIZK paradigm [AB19], while $\mathcal{F}_{\text{proof}}$ still enforces its rules.

Functionality $\mathcal{F}_{\text{CRS}}$

Initialisation:
  $(pk_{\text{CRS}}, _) \xleftarrow{\$} \text{KEYGEN}()$

On (GETCRS) by $P/\mathcal{E}/\mathcal{A}$:
  reply pk_{CRS}

Fig. 9: Common Reference String functionality $\mathcal{F}_{\text{CRS}}$.

$\mathcal{F}_{\text{proof}}$ handles proofs of correct construction of votes and aggregate shares.

Functionality $\mathcal{F}_{\text{proof}}$

On (PROVEVOTE, v, g_0, pk_{CRS}, $\langle g_k^s, g_k^{\text{CRS}} \rangle_{k\in[n]}$, $\langle c_k^s, c_k^{\text{CRS}}, P_k \rangle_{k\in[n]}$, sk) by P:
  if sk is given as input and $\text{PK}(sk) = pk_{\text{CRS}}$ **then**
    store $(P, \langle c_k^s, c_k^{\text{CRS}}, P_k \rangle_{k\in[n]}, pk_{\text{CRS}})$
    reply (OK)
  end if
  if $v \notin \{-1, 0, 1\}$ **then**
    reply (ERROR)
  end if
  $\langle s_k \rangle_{k\in[n]} \leftarrow \text{SHARE}(v; g_0)$
  for k from 1 to n **do**
    send (PK, P_k) to $\mathcal{G}_{\text{PKI}}$ and assign reply to pk_k
    $c^s \leftarrow \text{ENC}(s_k, pk_k; g_k^s)$
    $c^{\text{CRS}} \leftarrow \text{ENC}(s_k, pk_{\text{CRS}}; g_k^{\text{CRS}})$
    if $c^s \neq c_k^s \vee c^{\text{CRS}} \neq c_k^{\text{CRS}}$ **then**
      reply (ERROR)
    end if
  end for
  store $(P, \langle c_k^s, c_k^{\text{CRS}}, P_k \rangle_{k\in[n]}, pk_{\text{CRS}})$
  reply (OK)

On (VERIFYVOTE, $P, pk_{\text{CRS}}, \langle c_k^s, c_k^{\text{CRS}}, P_k \rangle_{k\in[n]}$) by $P/\mathcal{E}/\mathcal{A}$:
  if $(P, \langle c_k, c_k^{\text{CRS}}, P_k \rangle_{k\in[n]}, pk_{\text{CRS}})$ is stored **then** reply (OK) **else** reply (ERROR)

On (PROVESHARE, j, S, pk_{CRS}, sk) by P:
  if $\text{PK}(sk) = pk_{\text{CRS}}$ **then**
    store $(P, j, S, pk_{\text{CRS}})$
    reply (OK)
  end if
  send (VERIFY, sk, P) to $\mathcal{G}_{\text{PKI}}$ and assign reply to T

28: **if** $T = (\text{ERROR})$ **then** reply (ERROR)
29: send (READ) to $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$, keep $(\text{VOTE}, (r, \langle c_k, P_k \rangle_{k\in[n]}))$ entries from reply and collect them into $\texttt{votes} = \langle\{\mathsf{r} : r_i, \mathsf{c} : \langle c_{i,k}, P_{i,k}\rangle_{k\in[n]}\}\rangle_{i\in[R]}$
30: $\mathcal{B} = \langle\{\mathsf{r} : \langle r_i\rangle_{i\in[B]}, \mathsf{P} : \langle P_k\rangle\}\rangle \leftarrow \mathsf{Batch}(\texttt{votes})$
31: **if** any entry in votes with round $r \in \mathcal{B}_j.\mathsf{r}$ contains no ciphertext for P **then** reply (ERROR)
32: $C \leftarrow \sum_{i\in[|B|]} c : (\mathcal{B}_j.\mathsf{r}_i, \langle \ldots, (c, P), \ldots\rangle) \in \texttt{votes}$
33: **if** $S = \text{DEC}(C, sk)$ **then**
34: store $(P, j, S, pk_{\text{CRS}})$
35: reply (OK)
36: **else**
37: reply (ERROR)
38: **end if**

39: On $(\text{VERIFYSHARE}, P, pk_{\text{CRS}}, j, S)$ by $P/\mathcal{E}/\mathcal{A}$:
40: **if** $(P, j, S, pk_{\text{CRS}})$ is stored **then** reply (OK) **else** reply (ERROR)

Fig. 10: Zero-knowledge functionality $\mathcal{F}_{\text{proof}}$.

The $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ functionality provides the core functions regarding handling vote and aggregate share submissions. When a player asks to vote, it samples decryptors (using $\mathsf{getDecryptors}_{n,B,\mathcal{P}}()$), checks the validity of the vote and stores it. Similarly, when a player asks to submit an aggregate share, $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ verifies and stores it.

Functionality $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$

Initialisation:
  $\texttt{voted} \leftarrow \emptyset$
  $\mathcal{V} \leftarrow \emptyset$
  $\texttt{lastVoteSucceeded} \leftarrow \texttt{true}$
  send (GETCRS) to $\mathcal{F}_{\text{CRS}}$ and assign reply to pk_{CRS}
  for $P \in \mathcal{P}$ **do**
    send (PK, P) to $\mathcal{G}_{\text{PKI}}$ and assign reply to pk
    if pk is not a valid public key **then** halt
  end for

On (GETDECRYPTORS) by $P \in \mathcal{P}$:
  if $P \in \texttt{voted}$ **then** yield execution token
  add P to voted
  if lastVoteSucceeded **then**
    $\texttt{decryptors} \leftarrow \mathsf{getDecryptors}_{n,B,\mathcal{P}}()$
    parse decryptors as $(r, \langle P_i\rangle_{i\in[n]})$
  end if // else reuse previous decryptors

17: `lastVoteSucceeded` ← `false`
18: reply `decryptors` to P and expect reply (VOTE, r, $\langle c_i^s, c_i^{\text{CRS}}, P_i \rangle_{i \in [n]}$) by P // received r and $\langle P_i \rangle$ must match the ones stored in `decryptors`
19: send (VERIFYVOTE, P, pk_{CRS}, $\langle c_i^s, c_i^{\text{CRS}}, P_i \rangle_{i \in [n]}$) to $\mathcal{F}_{\text{proof}}$ and ensure reply is (OK)
20: `lastVoteSucceeded` ← `true`
21: add (VOTE, r, $\langle c_i^s, P_i \rangle_{i \in [n]}$) to $\mathcal{V}$

22: On (READ) by $P/\mathcal{E}/\mathcal{A}$:
23: reply $\mathcal{V}$

24: On (BATCHSHARE, j, S) by $P \in \mathcal{P}$:
25: send (VERIFYSHARE, P, pk_{CRS}, j, S) to $\mathcal{F}_{\text{proof}}$ and ensure reply is (OK)
26: add (BATCHSHARE, j, S) to $\mathcal{V}$

Fig. 11: $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ functionality: vote and share aggregator

The $\mathcal{F}_{\text{vote}}^{B,n,t}$ functionality serves the same functions as $\Pi_{\text{vote}}^{B,n,t}$ (Fig. 13), but aggregates locally the votes of honest players. It expects specific messages from the adversary which let it know when a corrupted player votes or drains a batch.

Functionality $\mathcal{F}_{\text{vote}}^{B,n,t}$

1: Initialisation:
2: `results` ← 0
3: `votes` ← ∅
4: `maliciousEntries` ← ∅

5: On M by corrupted P:
6: send (FORWARD, P, M) to $\mathcal{A}$

7: On (FORWARD, P, M) by $\mathcal{A}$ where P is corrupted:
8: send M to P

9: On first (INIT) by P: // one INIT per player
10: send (INIT, P) to $\mathcal{A}$ and assign reply to pk_P
11: reply pk_P to P

12: On (READ) by P:
13: reply (RESULTS, `results`)

14: On first (VOTE, v) by P:
15: send (GETDECRYPTORS, P) to $\mathcal{A}$, expect reply $(r, \langle P_k \rangle_{k \in [n]})$ by $\mathcal{A}$
16: **if** r is in `votes` or `maliciousEntries` **then** halt

17: **if** at least t players in $\langle P_k \rangle_{k\in[n]}$ are malicious **then** halt
18: append $(r, P, v, \langle P_k \rangle_{k\in[n]})$ to `votes`.
19: send $(\text{VOTED}, P, \langle P_k \rangle_{k\in[n]})$ to $\mathcal{A}$ // v is the only thing $\mathcal{F}_{\text{vote}}^{B,n,t}$ keeps private from $\mathcal{A}$

20: On $(\text{VOTED}, r, \langle P_k \rangle_{k\in[n]})$ by $\mathcal{A}$:
21: **if** r is in `votes` or in `maliciousEntries` **then** halt
22: add $\{\mathsf{r} : r, \mathsf{c} : \langle \bot, P_k \rangle_{k\in[n]}\}$ to `maliciousEntries`

23: On $(\text{DRAINED}, j, P)$ by $\mathcal{A}$ where P is corrupted:
24: $\mathsf{HasDrained}(P, \mathcal{B}_j) \leftarrow \mathsf{True}$
25: **if** there are exactly t distinct $P_i : \mathsf{HasDrained}(P_i, \mathcal{B}_j) = \mathsf{True}$ **then**
26: `newVotes` $\leftarrow \sum_{i\in[|B|]} v : (\mathcal{B}_j.\mathsf{r}_i, v, \cdot) \in$ `votes`
27: send $(\text{GETMALICIOUSVOTES}, j)$ to $\mathcal{A}$ and expect reply `maliciousVotes`
28: `results` $\leftarrow$ `results` + `newVotes` + `maliciousVotes`
29: **end if**
30: reply (OK)

31: On $(\text{GETHONESTVOTES}, j)$ by $\mathcal{A}$:
32: send (READ) to $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$, keep $(\text{VOTE}, _)$ entries from reply and collect them into `castVotes`
33: **if** the j-th batch in `castVotes` is complete **then**
34: send $(\text{HONESTVOTES}, j, \sum_{i\in[|B|]} v : (\mathcal{B}_j.\mathsf{r}_i, v, \cdot) \in \texttt{votes})$ to $\mathcal{A}$ // only disclose aggregates of complete batches
35: **end if**

36: On (DRAINBATCH) by P:
37: `entries` $\leftarrow$ `votes`.$\mathsf{map}((r, v, \langle P_k \rangle_{k\in[n]}) \mapsto \{\mathsf{r} : r, \mathsf{c} : \langle \bot, P_k \rangle\}) \cup$ `maliciousEntries`
38: $\mathcal{B} = \langle \{\mathsf{r} : \langle r_i \rangle_{i\in[B]}, \mathsf{P} : \langle P_j \rangle\} \rangle \leftarrow \mathsf{Batch}(\texttt{entries})$
39: $j \leftarrow \arg\min_{i\in[|\mathcal{B}|]} \{P \in \mathcal{B}_i.\mathsf{P} \wedge \mathsf{HasDrained}(P, \mathcal{B}_i) = \mathsf{False}\}$
40: **if** j exists **then** $\mathsf{HasDrained}(P, \mathcal{B}_j) \leftarrow \mathsf{True}$ **else return**
41: **if** there are exactly t distinct $P_i : \mathsf{HasDrained}(P_i, \mathcal{B}_j) = \mathsf{True}$ **then**
42: `newVotes` $\leftarrow \sum_{i\in[|B|]} v : (\mathcal{B}_j.\mathsf{r}_i, v, \cdot) \in$ `votes`
43: send $(\text{GETMALICIOUSVOTES}, j)$ to $\mathcal{A}$ and expect reply `maliciousVotes`
44: `results` $\leftarrow$ `results` + `newVotes` + `maliciousVotes`
45: **end if**
46: send $(\text{BATCHSHARE}, P, j)$ to $\mathcal{A}$

Fig. 12: Ideal functionality $\mathcal{F}_{\text{vote}}^{B,n,t}$, realised by $\Pi_{\text{vote}}^{B,n,t}$.

9 Formal Protocol $\Pi_{\text{vote}}^{B,n,t}$

Protocol $\Pi_{\text{vote}}^{B,n,t}$ is executed by each protocol party. At any time, $\mathcal{E}$ can ask a party to vote, drain the next available batch, or read decrypted results.

Protocol $\Pi_{\text{vote}}^{B,n,t}$ (self is P)

On first (INIT) by $\mathcal{E}$:
  send (GETCRS) to $\mathcal{F}_{\text{CRS}}$ and assign reply to pk_{CRS}
  send (INIT) to $\mathcal{G}_{\text{PKI}}$ and assign reply to pk
  reply pk to $\mathcal{E}$

On (READ) by $\mathcal{E}$:
  send (READ) to $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$, keep (BATCHSHARE, j, S) entries from reply and collect them into $\texttt{shares} = \langle j, S \rangle$
  $\texttt{results} \leftarrow 0$
  for i from 1 to $\max\{\texttt{shares.map}((j, S) \mapsto j)\}$ **do** // go over each batch
    if there are t (i, S) entries in `shares` **then**
      assign their shares to $(S_1, \ldots, S_t)$
      $\texttt{newVotes} \leftarrow \text{RECONSTRUCT}(S_1, \ldots, S_t)$
      $\texttt{results} \leftarrow \texttt{results} + \texttt{newVotes}$
    end if
  end for
  reply `results` to $\mathcal{E}$

On first (VOTE, v) by $\mathcal{E}$:
  ensure $v \in \{-1, 0, 1\}$, otherwise ignore
  send (GETDECRYPTORS) to $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ and assign reply to $(r, \langle P_k \rangle_{k \in [n]})$
  $g_0 \xleftarrow{\$} \mathcal{R}_{\text{share}}$
  $\langle s_k \rangle_{k \in [n]} \leftarrow \text{SHARE}(v; g_0)$
  for k in 1 to n **do**
    send (PK, P_k) to $\mathcal{G}_{\text{PKI}}$ and assign reply to pk_k
    $g_k^s \xleftarrow{\$} \mathcal{R}_{\text{enc}}$
    $c_k^s \leftarrow \text{ENC}(s_k, pk_k; g_k^s)$
    $g_k^{\text{CRS}} \xleftarrow{\$} \mathcal{R}_{\text{enc}}$
    $c_k^{\text{CRS}} \leftarrow \text{ENC}(s_k, pk_{\text{CRS}}; g_k^{\text{CRS}})$
  end for
  send (PROVEVOTE, v, g_0, pk_{CRS}, $\langle g_k^s, g_k^{\text{CRS}} \rangle_{k \in [n]}$, $\langle c_k^s, c_k^{\text{CRS}}, P_k \rangle_{k \in [n]}$) to $\mathcal{F}_{\text{proof}}$ and expect reply (OK)
  send (VOTE, r, $\langle c_k^s, c_k^{\text{CRS}}, P_k \rangle_{k \in [n]}$) to $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$

On (DRAINBATCH) by $\mathcal{E}$:
  send (READ) to $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$, keep (VOTE, $(r, \langle c_k, P_k \rangle_{k \in [n]})$) entries from reply and collect them into $\texttt{votes} = \langle \{\mathsf{r} : r_i, \mathsf{c} : \langle c_{i,k}, P_{i,k} \rangle_{k \in [n]}\} \rangle_{i \in [R]}$

32: $\mathcal{B} = \langle\{\mathsf{r} : \langle r_i \rangle_{i \in [B]}, \mathsf{P} : \langle P_k \rangle\}\rangle \leftarrow \mathsf{Batch}(\texttt{votes})$
33: $j \leftarrow \arg\min_{i \in [|\mathcal{B}|]} \{P \in \mathcal{B}_i.\mathsf{P} \wedge \mathsf{Drained}(\mathcal{B}_i) = \mathsf{False}\}$
34: **if** j exists **then**
35: $\mathsf{Drained}(\mathcal{B}_j) \leftarrow \mathsf{True}$
36: $C \leftarrow \sum_{i \in [|\mathcal{B}|]} c : (\mathcal{B}_j.\mathsf{r}_i, \langle \ldots, (c, P), \ldots \rangle) \in \texttt{votes}$
// $\forall r \in \mathcal{B}_j.\mathsf{r}, \exists (c, P)$ since $P \in \mathcal{B}_j.\mathsf{P}$
37: send (SK) to $\mathcal{G}_{\text{PKI}}$ and assign reply to sk
38: $S \leftarrow \text{DEC}(C, sk)$
39: send (PROVESHARE, j, S, pk_{CRS}, sk) to $\mathcal{F}_{\text{proof}}$ and expect reply (OK)
40: send (BATCHSHARE, j, S) to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$
41: **end if**

Fig. 13: $\Pi^{B,n,t}_{\text{vote}}$ protocol: executed by honest parties of our opinion aggregation scheme

10 Future Work

A number of future directions are discussed next. One possible enhancement to the scheme would be to instantiate a single, efficient protocol for multi-topic continuous opinion aggregation, avoiding the need to execute a separate copy of the protocol per topic. To achieve this improvement, a way to extend the additive-homomorphic properties of the secret sharing and the encryption schemes from a single, scalar integer to a vector of integers would be needed. A simple trick would be to multiply the vote of each topic with an increasing power of 2, such that each topic has enough bits between its power of 2 and that of the next topic to represent all practically plausible results, and add the votes together to form a single, albeit longer, integer. This would however introduce the limitation that each party would need to vote for all topics at once. Lifting this limitation, possibly employing a range proof for each vote to ensure its topic does not clash with that of a previously submitted vote, would be of interest.

Two further goals regarding the number and availability of players are the following: Firstly permitting dynamic participation, i.e., not requiring the set of all players to be known from the onset of the protocol. In its current form, our protocol can support social media groups with a closed member set — lifting this limitation would make it suitable for general social media platform use. Secondly allowing parties to vote again in case their batch is not drained by at least t decryptors after a set amount of time, invalidating their old vote in the process, thus further enhancing liveness. Achieving these two targets while maintaining good performance properties may also require employing a reputation system that rewards well-behaving, available parties and punishes misbehaving ones.

Furthermore, the security model can be made more robust. There are two orthogonal directions: on the one hand fully adaptive corruptions could be supported, which would likely need to make the order of votes and/or the corresponding sets of decryptors less predictable, or use differential privacy. On the other hand, proving security in a framework that supports composition, such as Universal Composition [Ca01] or Constructive Cryptography [Ma11], would enable composing this protocol with other, independent protocols that either run concurrently in the same machines or use our protocol as a subroutine.

Last but not least, the protocol now requires $O(|\mathcal{P}|)$ per-party resources, hindering scalability for large $|\mathcal{P}|$. We leave improving on this as future work.

11 Conclusion

In this work we defined a novel decentralised voting scheme, as well as formally proved its privacy and timely progress. We leveraged various primitives to create a protocol that enables a number of parties to express their opinion on a particular topic while maintaining their privacy within an anonymity set of fixed size B. Votes are revealed in batches that only disclose a single number, their sum. Our simulations showed that tuning the parameters allows for a variety of tradeoffs between, among others, the speed of batch revelation and the size of the anonymity sets. We also provided an implementation in Solidity.

Acknowledgements

This work was partly supported by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

This work was partly carried out at Imperial College London.

References

[18] Steem Whitepaper, 2018, https://steem.io/steem-whitepaper.pdf, accessed: 07/29/2026.

[20] Farcaster, 2020, https://www.farcaster.xyz/, accessed: 07/29/2026.

[22] Lens, 2022, https://www.lens.xyz/, accessed: 07/29/2026.

[AB19] Atapoor, S.; Baghery, K.: Simulation Extractability in Groth's zk-SNARK. In (Pérez-Solà, C. et al., eds.): Data Privacy Management, Cryptocurrencies and Blockchain Technology. Springer International Publishing, Cham, pp. 336–354, 2019.

[Ad08] Adida, B.: Helios: Web-based Open-Audit Voting. In: 17th USENIX Security Symposium. USENIX Association, San Jose, CA, USA, pp. 335–348, 2008.

[AHM14] Abbassi, Z.; Hegde, N.; Massoulié, L.: Distributed content curation on the Web. ACM Transactions on Internet Technology (TOIT) 14 (2-3), p. 9, 2014.

[Ba20] Baldimtsi, F. et al.: Crowd Verifiable Zero-Knowledge and End-to-End Verifiable Multiparty Computation. In: Advances in Cryptology - 26th ASIACRYPT. Springer, Daejeon, South Korea, pp. 717–748, 2020.

[BCT21] Banerjee, A.; Clear, M.; Tewari, H.: zkHawk: Practical Private Smart Contracts from MPC-based Hawk. In: 2021 3rd Conference on Blockchain Research Applications for Innovative Networks and Services (BRAINS). IEEE, no address given, pp. 245–248, 2021.

[Be11] Bergstrom, K.: "Don't feed the troll": Shutting down debate about community expectations on Reddit.com. First Monday 16 (8), 2011.

[Be13] Benaloh, J. et al.: STAR-Vote: A Secure, Transparent, Auditable, and Reliable Voting System. In: 2013 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '13, Washington, D.C., USA, August 12-13, 2013. 2013, https://www.usenix.org/conference/evtwote13/workshop-program/presentation/bell.

[Be14] Ben-Sasson, E. et al.: Zerocash: Decentralized Anonymous Payments from Bitcoin, Cryptology ePrint Archive, Report 2014/349, 2014.

[Be23] Bellés-Muñoz, M. et al.: Circom: A Circuit Description Language for Building Zero-Knowledge Applications. IEEE Transactions on Dependable and Secure Computing 20 (6), pp. 4733–4751, 2023.

[Bo20] Bowe, S. et al.: ZEXE: Enabling Decentralized Private Computation. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, no address given, pp. 947–964, 2020.

[BT94] Benaloh, J.; Tuinstra, D.: Receipt-free secret-ballot elections (extended abstract). In: Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing. STOC '94, Association for Computing Machinery, Montreal, Quebec, Canada, pp. 544–553, 1994, https://doi.org/10.1145/195058.195407.

[Bü20] Bünz, B. et al.: Zether: Towards Privacy in a Smart Contract World. In (Bonneau, J.; Heninger, N., eds.): Financial Cryptography and Data Security. Springer International Publishing, Cham, pp. 423–443, 2020.

[BY86] Benaloh, J. C.; Yung, M.: Distributing the power of a government to enhance the privacy of voters. In: Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing. PODC '86, Association for Computing Machinery, Calgary, Alberta, Canada, pp. 52–62, 1986, https://doi.org/10.1145/10590.10595.

[Ca01] Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. In: 42nd Annual Symposium on Foundations of Computer Science. IEEE, Las Vegas, Nevada, USA, pp. 136–145, 2001.

[CGS97] Cramer, R.; Gennaro, R.; Schoenmakers, B.: A Secure and Optimally Efficient Multi-Authority Election Scheme. In (Fumy, W., ed.): Advances in Cryptology — EUROCRYPT '97. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 103–118, 1997.

[CGT18] Cortier, V.; Galindo, D.; Turuani, M.: A Formal Analysis of the Neuchatel e-Voting Protocol. In: IEEE European Symposium on Security and Privacy. IEEE Computer Society, London, United Kingdom, pp. 430–442, 2018.

[Ch16] Chondros, N. et al.: D-DEMOS: A Distributed, End-to-End Verifiable, Internet Voting System. In: 36th International Conference on Distributed Computing Systems. IEEE Computer Society, Nara, Japan, pp. 711–720, 2016.

[Ch19] Chakravarty, M. M. T. et al.: Functional blockchain contracts, https://iohk.io/en/research/library/papers/functional-blockchain-contracts/, 2019.

[Co16] Cortier, V. et al.: SoK: Verifiability Notions for E-Voting Protocols. In: IEEE Symposium on Security and Privacy. IEEE Computer Society, San Jose, CA, USA, pp. 779–798, 2016.

[Cr20] Crites, E. C. et al.: Reputable List Curation from Decentralized Voting. Proc. Priv. Enhancing Technol. 2020 (4), pp. 297–320, 2020.

[CRS05] Chaum, D.; Ryan, P. Y. A.; Schneider, S. A.: A Practical Voter-Verifiable Election Scheme. In: Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings. Pp. 118–139, 2005.

[Cu15] Culnane, C. et al.: vVote: A Verifiable Voting System. ACM Trans. Inf. Syst. Secur. 18 (1), 3:1–3:30, 2015.

[Da10] Das Sarma, A. et al.: Ranking mechanisms in twitter-like forums. In: Proceedings of the third ACM international conference on Web search and data mining. ACM, ACM, New York, NY, USA, pp. 21–30, 2010.

[Da12] Damgård, I. et al.: Multiparty Computation from Somewhat Homomorphic Encryption. In: Advances in Cryptology - 32nd CRYPTO. Springer, Santa Barbara, CA, USA, pp. 643–662, 2012.

[Di03] Dini, G.: A secure and available electronic voting service for a large-scale distributed system. Future Gener. Comput. Syst. 19 (1), pp. 69–85, 2003.

[El85] Elgamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Transactions on Information Theory 31 (4), pp. 469–472, 1985.

[FOO92] Fujioka, A.; Okamoto, T.; Ohta, K.: A Practical Secret Voting Scheme for Large Scale Elections. In: AUSCRYPT '92, Workshop on the Theory and Application of Cryptographic Techniques. Springer, Gold Coast, Queensland, Australia, pp. 244–251, 1992.

[Ga24] Garg, S. et al.: hinTS: Threshold Signatures with Silent Setup. In: 2024 IEEE Symposium on Security and Privacy (SP). Pp. 3034–3052, 2024.

[Gr04] Groth, J.: Efficient maximal privacy in boardroom voting and anonymous broadcast. In: International Conference on Financial Cryptography. Springer, Berlin, Heidelberg, pp. 90–104, 2004.

[Gr16] Groth, J.: On the Size of Pairing-Based Non-interactive Arguments. In: Advances in Cryptology - 35th EUROCRYPT. Springer, Vienna, Austria, pp. 305–326, 2016.

[Ha20] Haenni, R. et al.: CHVote: Sixteen Best Practices and Lessons Learned. In: Electronic Voting - 5th International Joint Conference, E-Vote-ID 2020, Bregenz, Austria, October 6-9, 2020, Proceedings. Springer, no address given, pp. 95–111, 2020.

[HYL20] Han, R.; Yu, J.; Lin, H.: RandChain: Decentralised Randomness Beacon from Sequential Proof-of-Work. IACR Cryptol. ePrint Arch. 2020, p. 1033, 2020, https://eprint.iacr.org/2020/1033.

[JCJ05] Juels, A.; Catalano, D.; Jakobsson, M.: Coercion-resistant electronic elections. In: 2005 ACM Workshop on Privacy in the Electronic Society. Association for Computing Machinery, Alexandria, VA, USA, pp. 61–70, 2005.

[JRF09] Joaquim, R.; Ribeiro, C.; Ferreira, P.: VeryVote: A Voter Verifiable Code Voting System. In: E-Voting and Identity, Second International Conference, VoteID 2009, Luxembourg, September 7-8, 2009. Proceedings. Springer, no address given, pp. 106–121, 2009.

[Ka18] Kalodner, H. et al.: Arbitrum: Scalable, private smart contracts. In: 27th USENIX Security Symposium (USENIX Security 18). USENIX Association, Baltimore, MD, pp. 1353–1370, 2018, https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner.

[Ki19] Kiayias, A. et al.: A Puff of Steem: Security Analysis of Decentralized Content Curation. In: International Conference on Blockchain Economics, Security and Protocols, Tokenomics 2019. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Paris, France, 3:1–3:21, 2019.

[KKK21] Kerber, T.; Kiayias, A.; Kohlweiss, M.: KACHINA - Foundations of Private Smart Contracts. In: 34th Computer Security Foundations Symposium. IEEE, Dubrovnik, Croatia, pp. 1–16, 2021.

[KSW05] Karlof, C.; Sastry, N.; Wagner, D. A.: Cryptographic Voting Protocols: A Systems Perspective. In: 14th USENIX Security Symposium. USENIX Association, Baltimore, MD, USA, 2005.

[Ku14] Kulyk, O. et al.: Electronic voting with fully distributed trust and maximized flexibility regarding ballot design. In: 6th International Conference on Electronic Voting: Verifying the Vote (EVOTE). IEEE, Lochau / Bregenz, Austria, pp. 1–10, 2014.

[KY02] Kiayias, A.; Yung, M.: Self-tallying Elections and Perfect Ballot Secrecy. In: Public Key Cryptography, 5th International Workshop on Practice and Theory in Public Key Cryptosystems. Springer, Berlin, Heidelberg, pp. 141–158, 2002.

[KZZ15] Kiayias, A.; Zacharias, T.; Zhang, B.: End-to-End Verifiable Elections in the Standard Model. In: Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II. Pp. 468–498, 2015.

[KZZ17] Kiayias, A.; Zacharias, T.; Zhang, B.: An Efficient E2E Verifiable E-voting System without Setup Assumptions. IEEE Secur. Priv. 15 (3), pp. 14–23, 2017.

[Li17] Lindell, Y.: How to Simulate It - A Tutorial on the Simulation Proof Technique. In: Tutorials on the Foundations of Cryptography. Vol. 24, Springer, Cham, pp. 277–346, 2017.

[Li19] Liu, D. et al.: Anonymous Reputation System for IIoT-Enabled Retail Marketing Atop PoS Blockchain. IEEE Transactions on Industrial Informatics 15 (6), pp. 3527–3537, 2019.

[Ma11] Maurer, U.: Constructive Cryptography - A New Paradigm for Security Definitions and Proofs. In: Theory of Security and Applications - Joint Workshop. Springer, Saarbrücken, Germany, pp. 33–56, 2011.

[MKS17] Mell, P.; Kelsey, J.; Shook, J. M.: Cryptocurrency Smart Contracts for Distributed Consensus of Public Randomness. In: Stabilization, Safety, and Security of Distributed Systems. Springer, Boston, MA, USA, pp. 410–425, 2017.

[Na08] Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[Pa99] Paillier, P.: Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In (Stern, J., ed.): Advances in Cryptology — EUROCRYPT '99. Springer, Berlin, Heidelberg, pp. 223–238, 1999.

[Pe91] Pedersen, T. P.: A threshold cryptosystem without a trusted party. In: Proceedings of the 10th Annual International Conference on Theory and Application of Cryptographic Techniques. EUROCRYPT'91, Springer-Verlag, Brighton, UK, pp. 522–526, 1991.

[PS17] Pass, R.; Shi, E.: Hybrid Consensus: Efficient Consensus in the Permissionless Model. In (Richa, A., ed.): 31st International Symposium on Distributed Computing (DISC 2017). Vol. 91. Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 39:1–39:16, 2017, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.DISC.2017.39.

[RRI16] Ryan, P. Y. A.; Rønne, P. B.; Iovino, V.: Selene: Voting with Transparent Verifiability and Coercion-Mitigation. In: Financial Cryptography and Data Security - Workshops, BITCOIN, VOTING, and WAHC. Springer, Christ Church, Barbados, pp. 176–192, 2016.

[Sc99] Schoenmakers, B.: A Simple Publicly Verifiable Secret Sharing Scheme and Its Application to Electronic Voting. In (Wiener, M., ed.): Advances in Cryptology — CRYPTO' 99. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 148–164, 1999.

[Sh79] Shamir, A.: How to Share a Secret. Commun. ACM 22 (11), pp. 612–613, 1979.

[Sk13] Skala, M.: Hypergeometric tail inequalities: ending the insanity, 2013, arXiv: 1311.5939 [math.PR].

[SSG12] Stanoevska-Slabeva, K.; Sacco, V.; Giardina, M.: Content Curation: a new form of gatewatching for social media? Documento electrónico. Recuperado el 16, pp. 1–33, 2012.

[St19] Steffen, S. et al.: Zkay: Specifying and Enforcing Data Privacy in Smart Contracts. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. CCS '19, Association for Computing Machinery, London, United Kingdom, pp. 1759–1776, 2019.

[Su17] Sun, S. et al.: RingCT 2.0: A Compact Accumulator-Based (Linkable Ring Signature) Protocol for Blockchain Cryptocurrency Monero. In: 22nd European Symposium on Research in Computer Security. Springer, Oslo, Norway, pp. 456–474, 2017.

[Wo14] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger, https://ethereum.github.io/yellowpaper/paper.pdf, 2014.

[Yu09] Yu, H. et al.: Dsybil: Optimal sybil-resistance for recommendation systems. In: 2009 30th IEEE Symposium on Security and Privacy. IEEE, IEEE Computer Society, Oakland, California, USA, pp. 283–298, 2009.

[ZNP15] Zyskind, G.; Nathan, O.; Pentland, A.: Enigma: Decentralized Computation Platform with Guaranteed Privacy. CoRR abs/1506.03471, 2015, arXiv: 1506.03471, http://arxiv.org/abs/1506.03471.

[ZOB19] Zhang, B.; Oliynykov, R.; Balogun, H.: A Treasury System for Cryptocurrencies: Enabling Better Collaborative Intelligence. In: 26th Annual Network and Distributed System Security Symposium. The Internet Society, San Diego, California, USA, 2019.

A Supporting Logic for Ideal World

The $\mathsf{Batch}_B()$ function is used internally by $\mathcal{F}_{\text{proof}}$, the protocol $\Pi_{\text{vote}}^{B,n,t}$ and the functionality $\mathcal{F}_{\text{vote}}^{B,n,t}$. It groups votes into valid batches in a deterministic manner. In Appx. B we present a generalized version of Batch, suitable if any arbitrary method of choosing decryptors instead of the current $\mathsf{getDecryptors}_{n,B,\mathcal{P}}()$ were used by $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$.

Algorithm $\mathsf{Batch}_B(\mathtt{votes})$

$\mathcal{B} \leftarrow \langle\rangle$ // vector
for i from 0 to $\lfloor \frac{|\mathtt{votes}|}{B} \rfloor$ **do** // including 0, not including $\lfloor \frac{|\mathtt{votes}|}{B} \rfloor$
    $\mathtt{rs} \leftarrow \langle i \cdot B + 1, \ldots, (i+1) \cdot B \rangle$ // set rounds to vector from $i \cdot B + 1$ to $(i+1) \cdot B$ (inclusive), as votes stored by $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ have sequential rounds
    $\mathtt{Ps} \leftarrow \mathtt{votes}[i \cdot B + 1].\mathsf{c.map}((_, P) \mapsto P)$ // every B consecutive votes stored by $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ have the same decryptors
    append $(\mathtt{rs}, \mathtt{Ps})$ to $\mathcal{B}$
end for
return $\mathcal{B}$

Fig. 14: Batch_B function: groups votes into batches. Used by $\mathcal{F}_{\text{proof}}$, $\Pi^{B,n,t}_{\text{vote}}$, and $\mathcal{F}^{B,n,t}_{\text{vote}}$.

Key management is governed by the $\mathcal{G}_{\text{PKI}}$ functionality.

Functionality $\mathcal{G}_{\text{PKI}}$

On (INIT) by P:
    ensure this is the first time we receive (INIT) by P
    $(pk_P, sk_P) \xleftarrow{\$} \text{KEYGEN}()$
    reply (pk_P)

On (PK, P) by $P/\mathcal{E}/\mathcal{A}$:
    reply (pk_P)

On (SK) by P:
    reply (sk_P)

On (VERIFY, sk, P) by $P/\mathcal{E}/\mathcal{A}$ by $P/\mathcal{E}/\mathcal{A}$:
    if $sk = sk_P$ **then** reply (OK) **else** reply (ERROR)

Fig. 15: Key management functionality $\mathcal{G}_{\text{PKI}}$.

B Generalised Batch function

The generalized Batch algorithm forms batches using the oldest votes first, adding the oldest vote not considered yet in each new attempt to form a batch. In the general case in

which decryptors are chosen in an arbitrary fashion for each vote (instead of keeping the same decryptors for B consecutive votes), the DRAINBATCH function needs to deal with the possibility that several eligible batches might appear simultaneously. In such a case, it is important to break ties in a deterministic manner so that every party receives the same batches — this implementation (Fig. 16) chooses a lexicographic ordering (l. 6). Otherwise, whenever there is exactly one eligible batch, it simply chooses that.

Algorithm Generalised $\mathsf{Batch}_{B,t}(\texttt{votes})$

1: $\mathcal{B} \leftarrow \langle\rangle$ // vector
2: **for** top from B to $\max\{\texttt{votes}.\mathsf{map}((r,\cdot) \mapsto r)\}$ **do**
3: $\quad\texttt{entries_up_to_top} \leftarrow \texttt{votes}.\mathsf{filter}((r,\cdot) \mapsto r \leq \texttt{top})$
4: $\quad$// a valid batch consists of a vector of rounds $\mathsf{r} = \langle r_i \in [\texttt{top}]\rangle_{i\in[B]}$ (B natural numbers up to top) and a vector of parties $\mathsf{P} = \langle P_i \in \mathcal{P}\rangle_{i\in[m]}$ of length $t \leq m \leq n$ such that P_i is in every vote of $\texttt{votes}.\mathsf{filter}(r \in \mathsf{r})$. If some $P \in \mathcal{P}$ appears multiple times in P, it must also be in each each vote of interest at least as many times.
$(\forall i \in [m], \forall \texttt{vote} \in \texttt{votes}.\mathsf{filter}(r \in \mathsf{r}), |\texttt{vote}.\mathsf{filter}(P = P_i)|) \geq |\mathsf{P}.\mathsf{filter}(P = P_i)|)$
5: $\quad$**if** $\exists$ a valid batch $\{\mathsf{r} : \langle r_i\rangle_{i\in[B]}, \mathsf{P} : \langle P_i\rangle_{i\in[m]}\}$ in $\texttt{entries_up_to_top}$ **then**
6: $\quad\quad$append the batch to $\mathcal{B}$; if there are more than one valid batches, choose the minimum, compared based on the lexicographic ordering of the concatenation of the identifiers of all involved parties, $P_1P_2\ldots P_m$. // It is impossible that exactly the same parties are involved in two different batches, as that would mean that the batches' difference is in their entries' r. In that case each of the two batches would have $B-1$ entries with $r \neq \texttt{top}$ with at least one difference in these entries. This means that there would be at least B distinct entries with $r \neq \texttt{top}$ and t common parties. These entries would however form a batch D without the entry that has $r = \texttt{top}$. If $\texttt{top} = B$, the exclusion of the entry with $r = B$ would mean that there are not enough entries to form any batch, else D would have been already consumed in a previous iteration.
7: $\quad\quad$remove every entry involved in the new batch from $\texttt{entries_up_to_top}$ // This removal invalidates all alternative batches of the line above, therefore asserting the argument that there cannot be two or more valid, disjoint batches per iteration.
8: $\quad$**end if**
9: **end for**
10: **return** $\mathcal{B}$

Fig. 16: General version of the $\mathsf{Batch}_{B,t}$ function (cf. Fig. 14). Compatible with arbitrary decryptor choice methods that could be used by $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ instead of $\mathsf{getDecryptors}_{n,B,\mathcal{P}}()$ (Fig. 8).

C Proofs for Security, Privacy & Liveness Guarantees

Here we provide proofs for the security theorems of Sec. 5.

Thm. 3.

Algorithm findShare$(S_1, \ldots, S_{t-1}, x_t, V)$

Finds $S_t = (x_t, y_t)$: RECONSTRUCT$(S_1, \ldots, S_t) = V$ and the source polynomial of the secret, $P(x)$. Works for Shamir's Secret Sharing scheme [Sh79].

1: **for** i from 1 to $t-1$ **do**
2: parse S_i as (x_i, y_i) // x_i, y_i are finite field elements
3: **end for**
4: Solve linear equation $V = \sum_{i=1}^{t} y_i \prod_{\substack{m=1 \\ m \neq j}}^{m=t} \frac{x_m}{x_m - x_i}$ to find the only unknown, y_t // y_t is a finite field element
5: **for** i from 1 to t **do**
6: $l_i(x) = \prod_{\substack{m=1 \\ m \neq i}}^{t} \frac{x - x_m}{x_i - x_m}$
7: **end for**
8: **return** $((x_t, y_t), \sum_{i=1}^{t} y_i l_i)$

Fig. 17

Simulator $\mathcal{S}$ – all messages by $\mathcal{F}_{\text{vote}}^{B,n,t}$, unless otherwise noted

$\mathcal{S}$ takes over the interface of $\mathcal{F}_{\text{CRS}}$ and uses pk_{CRS} in $\mathcal{F}_{\text{CRS}}$ as generated by $\mathcal{S}$ in l. 3.

1: Initialisation:
2: `drained` $\leftarrow (0, \ldots, 0)$ // 1 entry for each possible batch
$\lceil |\mathcal{P}|/B \rceil$
3: $(pk_{\text{CRS}}, sk_{\text{CRS}}) \xleftarrow{\$}$ KEYGEN()

4: On (FORWARD, P, M):
5: send M, addressed from P, to internal $\mathcal{A}$

6: On M for corrupted party P by $\mathcal{A}$:
7: **if** $M = (\text{VOTE}, r, \langle c_k^s, c_k^{\text{CRS}}, P_k \rangle_{k \in [n]})$ **then**
8: send $(\text{VERIFYVOTE}, P, pk_{\text{CRS}}, \langle c_k^s, c_k^{\text{CRS}}, P_k \rangle_{k \in [n]}))$ to $\mathcal{F}_{\text{proof}}$
9: **if** reply by $\mathcal{F}_{\text{proof}}$ is (OK) **then**
10: send $(\text{VOTED}, r, \langle P_k \rangle_{k \in [n]})$ to $\mathcal{F}_{\text{vote}}^{B,n,t}$ and expect reply (OK)
11: **if** this vote completes the j-th batch **then**
12: assign number of corrupted decryptors in j-th batch to $\texttt{drained}_j$
13: **end if**
14: $v_P \leftarrow \text{RECONSTRUCT}(\text{DEC}(sk_{\text{CRS}}, c_1^{\text{CRS}}), \ldots, \text{DEC}(sk_{\text{CRS}}, c_t^{\text{CRS}}))$
15: **end if**
16: **else if** $M = (\text{BATCHSHARE}, j, S)$ **then**
17: send $(\text{VERIFYSHARE}, P, pk_{\text{CRS}}, j, S)$ to $\mathcal{F}_{\text{proof}}$
18: **if** reply by $\mathcal{F}_{\text{proof}}$ is (OK) **then**
19: send $(\text{DRAINED}, j, P)$ to $\mathcal{F}_{\text{vote}}^{B,n,t}$ and expect reply (OK)

20: **end if**
21: **end if**
22: send (FORWARD, P, M) to $\mathcal{F}_{\text{vote}}^{B,n,t}$

23: On (INIT, P):
24: send (INIT) to $\mathcal{G}_{\text{PKI}}$ as P and assign reply to pk_P // "as P" messages are implicitly routed through $\mathcal{F}_{\text{vote}}^{B,n,t}$
25: reply pk_P to $\mathcal{F}_{\text{vote}}^{B,n,t}$

26: On (GETDECRYPTORS, P):
27: send (GETDECRYPTORS) to $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ as P and forward reply to $\mathcal{F}_{\text{vote}}^{B,n,t}$

28: On (VOTED, P, $\langle P_k \rangle_{k\in[n]}$):
29: **for** k from 1 to n **do**
30: send (PK, P_k) to $\mathcal{G}_{\text{PKI}}$ and assign reply to pk_k
31: $c_k^s \leftarrow \text{ENC}(pk_k, 0)$
32: $c_k^{\text{CRS}} \leftarrow \text{ENC}(pk_{\text{CRS}}, 0)$
33: send (PROVEVOTE, $\bot, \bot, \bot, \bot, \langle c_k^s, c_k^{\text{CRS}}, P_k \rangle_{k\in[n]}, sk_{\text{CRS}}$) to $\mathcal{F}_{\text{proof}}$ and expect reply (OK)
34: **end for**
35: **if** this vote completes the j-th batch **then**
36: assign number of corrupted decryptors in j-th batch to $\texttt{drained}_j$
37: **end if**
38: send (VOTE, r, $\langle c_k^s, c_k^{\text{CRS}}, P_k \rangle_{k\in[n]}$) to $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ as P

39: On (GETMALICIOUSVOTES, j):
40: send (READ) to $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ and keep votes from reply
41: calculate the j-th batch $\mathcal{B}_j$
42: reply with sum of votes v_P of all corrupted voters in $\mathcal{B}_j$

43: On (BATCHSHARE, P, j):
44: **if** all voters of the j-th batch are corrupted **then**
45: execute ll. 31-32 and 36-39 of Fig. 13 as P // copy actions of $\Pi_{\text{vote}}^{B,n,t}$
46: reply (BATCHSHARE, j, S) // use S as calculated in l. 45
47: **end if**
48: find index k of decryptor P in j-th batch
49: $\texttt{drained}_j \leftarrow \texttt{drained}_j + 1$
50: **if** $\texttt{drained}_j < t$ **then**
51: $s_{P,j} \xleftarrow{\$} \mathcal{R}_{\text{SSS}}$ // $\mathcal{R}_{\text{SSS}}$ is the finite field of Shamir's Secret Sharing
52: $S \leftarrow (k, s_{P,j})$
53: send (PROVESHARE, j, S, pk_{CRS}, sk_{CRS}) to $\mathcal{F}_{\text{proof}}$ as P and expect reply (OK)
54: reply (BATCHSHARE, j, S)
55: **else if** $\texttt{drained}_j = t$ **then**
56: send (GETHONESTVOTES, j) to $\mathcal{F}_{\text{vote}}^{B,n,t}$, add to the reply the sum of corrupted votes in the j-th batch (as stored in l. 14) and assign result to V_j

57: send (READ) to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$, from the reply extract the $t-1$ aggregate shares of the j-th batch and assign them to $S_1, \ldots, S_{t-1}$
58: $(S_t, Q_j) \leftarrow \mathsf{findShare}(S_1, \ldots, S_{t-1}, k, V_j)$
59: send (PROVESHARE, j, S_t, pk_{CRS}, sk_{CRS}) to $\mathcal{F}_{\text{proof}}$ as P and expect reply (OK)
60: reply (BATCHSHARE, j, S_t)
61: **else** // $\texttt{drained}_j > t$
62: $S \leftarrow (k, Q_j(k))$
63: send (PROVESHARE, j, S, pk_{CRS}, sk_{CRS}) to $\mathcal{F}_{\text{proof}}$ as P and expect reply (OK)
64: send (BATCHSHARE, j, S) to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ as P
65: **end if**

Fig. 18

The contents of messages that involve corrupted parties are fully controlled by the internal, simulated adversary $\mathcal{A}$. $\mathcal{S}$ only extracts some data from some of these messages (Fig. 18, l. 12) and reads information from $\mathcal{F}_{\text{proof}}$ without causing any change to it (Fig. 18, ll. 8 and 17). $\mathcal{S}$ also informs $\mathcal{F}^{B,n,t}_{\text{vote}}$ whenever a corrupted player successfully votes (Fig. 18, l. 10) or drains a certain batch (Fig. 18, l. 19), which is information that $\mathcal{F}^{B,n,t}_{\text{vote}}$ uses to decide when a batch is fully drained and therefore whether it can disclose the aggregate votes for this share to $\mathcal{S}$. As we will see later in the proof, this information is necessary to ensure indistinguishability and could anyway be obtained by having $\mathcal{F}^{B,n,t}_{\text{vote}}$ read $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ directly. Therefore the way $\mathcal{S}$ handles corrupted players' messages cannot provide a distinguishing advantage.

In the real world, an honest party P that receives (INIT) sends (INIT) to $\mathcal{G}_{\text{PKI}}$ and forwards its reply pk to $\mathcal{E}$. In the ideal world, the $\mathcal{F}^{B,n,t}_{\text{vote}}$ notifies $\mathcal{S}$ that P received (INIT). $\mathcal{S}$ in turn sends (INIT) as P to $\mathcal{G}_{\text{PKI}}$ and forwards its reply to $\mathcal{F}^{B,n,t}_{\text{vote}}$, which stores and forwards it to $\mathcal{E}$. Since in both cases the response is generated by $\mathcal{G}_{\text{PKI}}$ with the same input by the same player, the reply received by $\mathcal{E}$ is perfectly indistinguishable in the two cases.

In the real world, an honest party that receives (VOTE, v) first gets the decryptors from $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ and then shares its vote and encrypts each share for each decryptor and with the CRS public key, proves correct ciphertext generation to $\mathcal{F}_{\text{proof}}$ and submits the vote to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$. In the ideal world, an honest party that receives (VOTE, v) allows $\mathcal{S}$ to generate and send the vote to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$. $\mathcal{S}$ chooses decryptors by asking $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$, as in the real world. This means that $\mathcal{F}^{B,n,t}_{\text{vote}}$ will not trigger the halt of Fig. 12, l. 16, as the local variable `votes` of $\mathcal{F}^{B,n,t}_{\text{vote}}$ only contains honestly generated votes, the same holds for VOTED messages by $\mathcal{S}$ and every honestly generated vote has a unique round. The same argument precludes the triggering of the halt of Fig. 12, l. 21. Also the halt of Fig. 12, l. 17 will only trigger with negligible probability by virtue of Thm. 1. Since it doesn't learn the plaintext vote, $\mathcal{S}$ then generates ciphertexts of 0 and proceeds to fake a proof of correctness of these, exploiting its knowledge of the secret corresponding to the CRS. It then sends the generated vote to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$. The round and decryptors of the vote are generated exactly as in the real world,

so they are perfectly indistinguishable. The ciphertexts' distribution is negligibly close to uniform independent of the input to ENC due to the IND-CPA property of the encryption scheme so the ciphertexts are indistinguishable. Lastly, in both worlds $\mathcal{F}_{\text{proof}}$ verifies the same statement, namely that the prover either knows the secret corresponding to the key that the prover provided as CRS (possible in the real world only if the prover maliciously sets its own key as CRS, which is irrelevant for honest verifiers who only accept proofs made with the pk_{CRS} given by $\mathcal{F}_{\text{CRS}}$) or that the prover has generated the ciphertexts honestly and in both worlds the proofs are correct. Therefore the state changes caused by a (VOTE, v) message to an honest party are indistinguishable between the two worlds.

In the real world, an honest party that receives (DRAINBATCH) reads all published votes, deterministically chooses a completed batch for which it is a decryptor (if any), sums the ciphertexts encrypted for itself, decrypts the sum to obtain one aggregate share, proves correct decryption and sends the share to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$. In the ideal world, when an honest party receives (DRAINBATCH), $\mathcal{F}^{B,n,t}_{\text{vote}}$ is aware of the same votes that it would know in the real world, since $\mathcal{F}^{B,n,t}_{\text{vote}}$ knows all honest votes by virtue of the dummy parties relaying them, as well as all malicious votes' rounds and decryptors since $\mathcal{S}$ sends a VOTED message with such information every time a malicious player successfully votes. $\mathcal{F}^{B,n,t}_{\text{vote}}$ can therefore proceed to calculate the same batches and deterministically find the same batch for the party to drain as in the real world, if any. In case this drain is the t-th for this batch, $\mathcal{F}^{B,n,t}_{\text{vote}}$ asks $\mathcal{S}$ for the sum of malicious votes for this batch and updates the total vote accordingly. It finally asks $\mathcal{S}$ to form a valid share, which in turn does so and sends it to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$.

$\mathcal{S}$ produces the share in the following fashion. In case all voters of this batch are malicious, they have produced all shares and ciphertexts correctly (otherwise their vote would have been ignored by $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$), so the ciphertexts correspond to the actual votes of the malicious players. In this case $\mathcal{S}$ follows the same steps that an honest decryptor would follow in the real world, therefore introducing no opportunity for distinguishability. In case however at least one voter is honest, $\mathcal{S}$ had faked its ciphertexts when the former voted. In order to generate the needed share now, $\mathcal{S}$ exploits the fact that it is free to decrypt the sum of the honest player's ciphertexts to any share, as long as its x component is equal to the index of the player in the batch's decryptors and fake a proof of correct decryption, by virtue of knowing the secret of the CRS. If the sum of total corrupted decryptors in this batch plus the number of honest decryptors that have drained this batch (including the current one) is less than t (call this number d), then the current drain will not contribute the t-th share for this batch, therefore it will not add anything to any party's existing knowledge. Also no party knows the value that the aggregate share would have in the real world, since at least one of the voters is honest and thus the shares that it encrypted for honest decryptors are never published. Note also that there is at least one honest decryptor in this batch, as $\mathcal{S}$ receives the current message, BATCHSHARE, by $\mathcal{F}^{B,n,t}_{\text{vote}}$ only when an honest decryptor drains a batch. Therefore $\mathcal{S}$ generates a random y value for the share, uses the secret part of the CRS to fake a proof of correct share and sends it to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$.

In case $d = t$, the share that will be created by $\mathcal{S}$ has to be such that reconstructing the t shares returns an aggregate vote matching that of the real world. To that end, $\mathcal{S}$ learns the sum of honest voters' votes in this batch by $\mathcal{F}_{\text{vote}}^{B,n,t}$ and adds the corrupted voters' votes in this batch to get the aggregate batch vote. Note that having $\mathcal{F}_{\text{vote}}^{B,n,t}$ disclose this sum does not leak any more information that what a malicious party in the real world could deduce after the aggregate share currently under generation would be made public. It then uses the aggregate vote and the $t - 1$ existing batch shares to solve a linear equation and obtain the y value of the new share, such that reconstruction leads to the correct value. It also calculates the polynomial that corresponds to the aggregate shares for future use (Fig. 17). It then fakes a proof of correct share generation and sends it to $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$. In this way, $\mathcal{S}$ has managed to only change data that cannot be known by malicious parties in the real world and keep all other data consistent between the two worlds.

In case $d > t$, $\mathcal{S}$ simply uses the previously determined polynomial to generate a new aggregate share that can be combined with the existing ones to correctly reconstruct the aggregate vote. It then fakes a proof of correct share generation and sends it to $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$.

In the real world, an honest party that receives (READ) reads all aggregate shares from $\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}$ and for each batch that has at least t such shares, the party reconstructs the aggregate vote and adds it to the sum. Once done, it returns the total. In the ideal world, $\mathcal{F}_{\text{vote}}^{B,n,t}$ simply returns the results as it has stored them locally. These two values necessarily coincide, since $\mathcal{F}_{\text{vote}}^{B,n,t}$ only updates the results once for each batch, exactly when the latter can be reconstructed for the first time. $\mathcal{F}_{\text{vote}}^{B,n,t}$ knows the honest votes of this batch, as it receives them from the honest dummy parties, and the correct malicious votes are given honestly by $\mathcal{S}$. Therefore the two worlds are indistinguishable. □

Thm. 1. In a real-world execution with parties in $\mathcal{P}$ of which at most s are malicious, let X be the event under which at least one of the sets of decryptors returned by $\mathsf{getDecryptors}_{n,B,\mathcal{P}}$ contains at least t malicious parties. We will show that

$$\begin{aligned}
&\forall \text{ PPT } \mathcal{A}, \mathcal{E}, \forall \text{ set of parties } \mathcal{P},\\
&\forall 0 < \alpha \leq 1 : \alpha|\mathcal{P}| \in \mathbb{N}, n = \alpha|\mathcal{P}|,\\
&\forall 0 < \gamma \leq 1 : \gamma n \in \mathbb{N}, t = \gamma n,\\
&\forall 0 \leq \beta < 1 : \beta|\mathcal{P}| \in \mathbb{N}, s = \beta|\mathcal{P}|, B \in \mathbb{N},\\
&\text{If } \frac{t}{n} > \frac{s}{|\mathcal{P}|} \text{ then } \Pr[X] \leq \text{negl}(n) \text{ in } \text{EXEC}^{\mathcal{G}_{\text{VoteBox}}^{\mathcal{P}}, \mathcal{F}_{\text{proof}}}_{\Pi_{\text{vote}}^{B,n,t}, \mathcal{A}, \mathcal{E}} .
\end{aligned}$$

$\mathsf{getDecryptors}_{n,B,\mathcal{P}}$ is called at most $|\mathcal{P}|$ times. The first time and every B times going forward, a new, uniformly random subset of n players is chosen, for a maximum total of $\lceil \frac{|\mathcal{P}|}{B} \rceil$ independent choices. Since decryptors are chosen after new corruptions for each batch, each random choice of n decryptors can be modelled as n consecutive single-ball draws without replacement and with replacement scheme $R(\text{white}) = R(\text{black}) = 0$ (i.e., after drawing any ball, no balls are added back to the urn) from an urn that initially has 1

white and 1 black ball for each honest and each malicious player respectively, for a total of s black and $|\mathcal{P}| - s$ white balls. The random variable b = "number of black balls drawn after n draws" follows a hypergeometric distribution [Sk13]. From the above we deduce that $\Pr[X] \leq \Pr[b \geq t]\lceil\frac{|\mathcal{P}|}{B}\rceil$. It is

$$E[b] = n\frac{s}{|\mathcal{P}|} \text{ and } \forall q \geq 0, \Pr[b \geq E[b] + qn] \leq e^{-2q^2n} \quad .$$

We want to upper bound $\Pr[b \geq t]$, therefore $t = E[b] + qn = n\frac{s}{|\mathcal{P}|} + qn \Leftrightarrow q = \frac{t}{n} - \frac{s}{|\mathcal{P}|}$, thus q is always positive due to the theorem prerequisite. It is

$$\Pr[b \geq t] \leq e^{-2(\frac{t}{n} - \frac{s}{|\mathcal{P}|})^2 n} = e^{-2(\gamma-\beta)^2 n} \quad . \tag{2}$$

Therefore $\Pr[X] \leq \lceil\frac{n}{\alpha B}\rceil e^{-2(\gamma-\beta)^2 n} \leq \text{negl}(n)$. □

Thm. 2. In a real-world execution, sending a GETDECRYPTORS and then a VOTE message to $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ is the only way for P to vote, whether P is honest or malicious (since $\mathcal{V}$, which contains all votes, is stored locally by $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$). $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ only accepts votes for which the decryptors have been generated by $\mathsf{getDecryptors}_{n,B,\mathcal{P}}$ (Fig. 11, ll. 14-18). $\mathsf{getDecryptors}_{n,B,\mathcal{P}}$ in turn keeps choosing the same decryptors until a batch of submitted votes is complete. In particular, a batch is complete every exactly B calls to $\mathsf{getDecryptors}_{n,B,\mathcal{P}}$, as in case a player does not cast its vote successfully (by either getting its decryptors but not submitting its vote before another player tries to vote, i.e. not providing the expected reply of Fig. 11, l. 18, or by submitting an invalid vote, i.e. having Fig. 11, l. 19 fail), $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$ reuses the same decryptors (as l. 20 of Fig. 11 was not run after l. 17 was run), therefore a party has to wait at most $B - 1$ successfully cast votes before its vote is included in a valid batch. □

D Smart Contract for $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$

Listing 1 is a practical implementation of the pseudocode of Fig. 11 in Solidity. In a production-grade implementation, the functions `verifyVoteShares` and `verifyBatchShare` (as well as the proof creation functions that would be run off-chain by voters and decryptors respectively to generate `zkProof`) would be generated by a ZK library (e.g., `circom` [Be23]) and the Solidity code would be heavily optimised.

```
pragma solidity >=0.8.26 <0.9.0;

contract VoteBox {

    mapping(address => Party) private parties;
    address[] private partyAddresses;
    address constant crs = 0xE0f5206BBD039e7b0592d8918820024e2a7437b9; // change to actual
        CRS
    uint constant decryptorsNo = 1000; // change to actual number of decryptors (n)
```

```
    uint constant batchSize = 20; // change to actual batch size (B)

    struct Party {
        bool exists;
        bool voted;
    }

    struct VoteShare {
        uint decryptorShare;
        uint crsShare;
        address decryptor;
    }

    struct Vote {
        VoteShare[decryptorsNo] voteShares;
    }

    struct VoteSecrets {
        int vote;
        bytes[] shareRand;
        bytes[][decryptorsNo] decryptorRand;
        bytes[][decryptorsNo] crsRand;
    }

    struct BatchShare {
        uint batchNo;
        bytes[] share;
    }

    struct Batch {
        uint[batchSize] rounds;
        address[decryptorsNo] decryptors;
    }

    Vote[] private votes;
    BatchShare[] private batchShares;
    bool lastVoteSucceeded = true;
    address currentVoter;
    uint private round = 0;
    uint private randomness = 0;
    mapping(address => bool) partyChosen;
    address[] private decryptors;

    constructor(address[] memory _parties) {
        for (
            uint i = 0;
            i < _parties.length;
            i++
        ) {
            parties[_parties[i]] = Party({
                exists: true,
                voted: false
            });
            partyAddresses.push(_parties[i]);
```

```
    }
}

function verifyVoteShares(
    VoteShare[decryptorsNo] calldata voteShares,
    bytes[] calldata zkProof
) private pure returns(bool) {
    // Verifies in zero knowledge that
    // `zkProof` guarantees that the
    // following logic returns true for
    // `voteShares`.

    // `VoteSecrets voteSecrets` is a
    // private input, concealed in
    // `zkProof`.

    // Verify vote is -1, 0, or 1.
    if (
        voteSecrets.vote != -1
        && voteSecrets.vote != 0
        && voteSecrets.vote != 1
    ) { return false; }

    // `share(vote, shareRand)` performs
    // secret sharing of `vote` with
    // randomness `shareRand`.
    bytes[decryptorsNo] expectedShares =
        share(
            voteSecrets.vote,
            voteSecrets.shareRand
        );
    for (
        uint i = 0;
        i < voteShares.length;
        i++
    ) {
        // `enc(share, decryptor, decrRand)`
        // encrypts `share` to public key
        // `decryptor` with randomness
        // `decrRand`.
        uint expectedDecryptorShare = enc(
          expectedShares[i],
          voteShares[i].decryptor,
          voteSecrets.decryptorRand[i]
        );
        // Verify that provided
        // `decryptorShare` has been
        // built correctly.
        if (
            expectedDecryptorShare
            != voteShares[i].decryptorShare
        ) { return false; }

        uint expectedCrsShare = enc(
```

```
                expectedShares[i],
                crs,
                voteSecrets.crsRand[i]
            );
            // Verify that provided crsShare has
            // been built correctly.
            if (
                expectedCrsShare
                != voteShares[i].crsShare
            ) { return false; }
        }
        return true;
    }

    function buildBatch() private view
        returns(Batch[] memory) {
        uint batchesNo = votes.length/batchSize;
        Batch[] memory res =
            new Batch[](batchesNo);

        for (uint i = 0; i < batchesNo; i++) {
            uint[batchSize] memory rounds;
            for (uint j=0; j < batchSize; j++) {
                rounds[j] = i*batchSize+1+j;
            }

            address[decryptorsNo] memory batchDecryptors;
            for (
                uint j = 0;
                j < decryptorsNo;
                j++
            ) {
                batchDecryptors[j] =
                votes[i*batchSize+1]
                  .voteShares[j]
                  .decryptor;
            }

            res[i] = Batch({
                rounds: rounds,
                decryptors: batchDecryptors
            });
        }
        return res;
    }

    function verifyBatchShare(
        BatchShare calldata share,
        bytes[] calldata zkProof
    ) private view returns(bool) {
        // Verifies in zero knowledge that
        // `zkProof` guarantees that the
        // following logic returns true
        // for `share`.
```

```
uint cipherSum = 0;
Batch memory batch =
    buildBatch()[share.batchNo];
for (
    uint i = batch.rounds[0];
    i < batch.rounds[batchSize];
    i++
) {
    bool found = false;
    for (
        uint j = 0;
        j < decryptorsNo;
        j++
    ) {
        if (
            votes[i]
                .voteShares[j]
                .decryptor
            == msg.sender
        ) {
            found = true;
            cipherSum += votes[i]
                .voteShares[j]
                .decryptorShare;
            break;
        }
    }
    // Verify that `msg.sender` has a
    // share in every vote in the batch.
    if (!found) {
        return false;
    }
}

// `uint sk` is a private input,
// concealed in `zkProof`.
// Build address from `sk`.
bytes memory pk = bytes.concat(
    keccak256((abi.encodePacked(sk)))
);
// Addresses span 20 bytes.
bytes memory bytesAddr = new bytes(20);
for (
    uint i = 0;
    i < pk.length - 12;
    i++
) {
    bytesAddr[i] = pk[i+12];
}

// Convert type `bytes` to `address`.
address addr;
assembly {
```

```
            addr := mload(add(bytesAddr, 20))
        }

        return
            // Verify that `sk` corresponds to
            // `msg.sender`.
            addr == msg.sender
            // Verify that the sum of
            // `msg.sender`'s  batch shares
            // decrypt to the input share.
            && share.share
            == decrypt(sk, cipherSum);
    }

    function getDecryptors() public
        returns(uint, address[] memory) {
        require(
            parties[msg.sender].exists,
            "Party not registered"
        );
        require(
            !parties[msg.sender].voted,
            "Party has already voted"
        );

        if (lastVoteSucceeded) {
            round += 1;

            if (round % batchSize == 1) {
                delete decryptors;

                // Build a uniformly random set
                // of `batchSize` decryptors.
                while (
                    decryptors.length < decryptorsNo
                ) {
                    // `random()` generates a
                    // uniformly random number
                    // from 0 to P-1.
                    uint idx = random(
                        0, partyAddresses.length
                    );

                    if (!partyChosen[
                        partyAddresses[idx]
                    ]) {
                        partyChosen[
                            partyAddresses[idx]
                        ] = true;
                        decryptors.push(
                            partyAddresses[idx]
                        );
                    }
                }
```

```
            // Clean `partyChosen` for
            // next use.
            for (
                uint i = 0;
                i < decryptors.length;
                i++
            ) {
                delete
                    partyChosen[
                        decryptors[i]
                    ];
            }
        }
    }

    lastVoteSucceeded = false;
    currentVoter = msg.sender;

    return (round, decryptors);
}

function vote(
    uint r,
    VoteShare[decryptorsNo] calldata voteShares,
    bytes[] calldata zkProof
) public {
    require(
        msg.sender == currentVoter
        && r == round
        && verifyVoteShares(voteShares, zkProof),
        "Invalid vote"
    );

    lastVoteSucceeded = true;

    votes.push();
    Vote storage temp =
        votes[votes.length - 1];
    for (uint i=0; i < decryptorsNo; i++) {
        temp.voteShares[i] = voteShares[i];
    }
}

function batchShare(
    BatchShare calldata share,
    bytes[] calldata zkProof
) public {
    require(
        parties[msg.sender].exists,
        "Party not registered"
    );
    require(
        verifyBatchShare(share, zkProof),
```

```
            "Invalid batch share"
        );

        batchShares.push(share);
    }

    function read() public view
        returns(
            Vote[] memory, BatchShare[] memory
        ) {
        return (votes, batchShares);
    }
}
```

List. 1: Smart Contract for $\mathcal{G}^{\mathcal{P}}_{\text{VoteBox}}$