hbACSS: How to Robustly Share Many Secrets

Thomas Yurek*, Licheng Luo*, Jaiden Fairoze†, Aniket Kate‡, Andrew Miller*
*University of Illinois at Urbana-Champaign, {*yurek2, ll6, soc1024*}*@illinois.edu*
†University of California, Berkeley, *fairoze@berkeley.edu*
‡Purdue University, *aniket@purdue.edu*

***Abstract*—Despite significant recent progress toward making multi-party computation (MPC) practical, no existing MPC library offers complete robustness—meaning guaranteed output delivery, including in the offline phase—in a network that even has intermittent delays. Importantly, several theoretical MPC constructions already ensure robustness in this setting. We observe that the key reason for this gap between theory and practice is the absence of efficient verifiable/complete secret sharing (VSS/CSS) constructions; existing CSS protocols either require a) challenging broadcast channels in practice or b) introducing computation and communication overhead that is at least quadratic in the number of players.**

This work presents hbACSS, a suite of optimal-resilience asynchronous complete secret sharing protocols that are (quasi)linear in both computation and communication overhead. Towards developing hbACSS, we develop hbPolyCommit, an efficient polynomial commitment scheme that is (quasi)linear (in the polynomial degree) in terms of computation and communication overhead without requiring a trusted setup. We implement our hbACSS protocols, extensively analyze their practicality, and observe that our protocols scale well with an increasing number of parties. In particular, we use hbACSS to generate MPC input masks: a useful primitive which had previously only been calculated nonrobustly in practice.

I. Introduction

Multiparty Computation (MPC) is a widely useful way to build confidentiality and privacy-preserving computations into a distributed system, with applications such as anonymous messaging [2], [3], [47], secure auctions and digital asset exchanges [26], [49], and parameter generation [28]. As such systems grow closer to practice, we also start to care about *robustness*, where secrecy and liveness guarantees hold in spite of intermittent (non-synchronous) networks and a minority of faulty nodes. While several MPC implementations and libraries [47], [5], [38], [59], [51], [43] have emerged over the last decade, it remains an open problem to provide practical robust MPC in networks without strong synchrony assumptions.

Recent work shows how to make the online phase of practical asynchronous MPC robust [47]. However, the preprocessing phase, which must be run by the servers prior to receiving input, is much harder to make robust in practice. To explain the problem we will focus on generating random input masks, which allow clients to easily contribute secret inputs to an MPC program. The goal is to produce a random secret sharing $[\![r]\!]_t$ which satisfies the following: if no more than t of the N MPC server nodes are corrupted, r is uniformly random, unknown, and any $t+1$ parties can reconstruct the same r.

The standard way to generate such values is for all the servers to contribute shares they sample individually, which are then all combined to extract fully random values, even if some corrupt parties chose their inputs in a correlated way [11]. This approach hinges on a protocol that can be used to verify that the individually chosen shares are chosen correctly. Verifiable Secret Sharing (VSS) [30], [40], [22] is a natural choice for this task. In particular, Asynchronous Complete Secret Sharing (ACSS) [54] provides all the robustness guarantees needed for the MPC application, since it guarantees not only that the secret inputs can be reconstructed if necessary, but also that each of the servers can receive its original share of the secret. However, existing ACSS protocols introduce a computation and communication overhead that is quadratic in the number of parties and cannot scale well beyond a small number of nodes. This motivates the design of an efficient, scalable ACSS protocol without compromising the optimal replication factor of $3t+1$ or the asynchronous communication setting. In targeting this threat model, we prefix the protocols designed in this paper with "hb" as a reference to the honey badger, a creature known for its resilience in harsh adversarial settings.

A. Challenges and overview of our solution

a) Good performance under worst case conditions: The asynchronous network setting is fundamentally challenging: unlike in synchrony, a protocol which waits to hear from all parties will stall indefinitely. Instead, we must proceed after hearing from only $N-t$ of the parties, where t is a bound on the number of parties that can fail. Since crashed nodes are indistinguishable from slow nodes, it could be that t parties for which we waited are corrupted, and thus only $N-2t$ correct parties received valid shares. To cope with asynchrony, the most closely related protocol, VSS-R [9], falls back to an inefficient backup mode with communication overhead that is quadratic in the number of servers, even after a brief period of desynchronization. Alternately, Patra et al. give AVSS protocols with linear communication overhead [53], but require weakening resilience ($t < N/4$). We remark, however, that many $t < N/3$, AVSS/AVCSS protocols (including ours) could improve their amortized bandwidth by a factor of $O(N)$ through the use of Packed Secret Sharing. Consequently, we focus on the $t < N/3$ setting, knowing that improvements here can also lead to improvements in more relaxed settings.

b) Aggressive batching for large secrets: Motivated by our application of MPC preprocessing, we seek general efficiency but assume the amount of data that needs to be shared is large. We focus on the case where each single dealer needs to deal a large batch of secrets, such as for precomputation purposes. In doing so, we can amortize away

Network and Distributed Systems Security (NDSS) Symposium 2022
27 February - 3 March 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.23120
www.ndss-symposium.org

one-time startup costs.

The secret share encodings and polynomial commitments in our work function similar to other VSS-based schemes. Starting from [40], cryptographic VSS protocols are centered around broadcasting a polynomial commitment [42] along with an evaluation proof that enables each server to validate the share they received. The performance challenge arises when providing enough redundancy for VSS to recover from missing or corrupt shares.

Our solution is based on what we call *encrypt-then-disperse*, which generalizes a technique from HoneyBadgerBFT [50]. Before transmitting the payload of secret shares and evaluation proofs, the payload is first encrypted using public-key encryption. Next, the encrypted payload is dispersed using an Asynchronous Verifiable Information Dispersal (AVID [23]) routine, which can be more efficient because we do not need to hide the already-encrypted payload. The use of AVID guarantees that every honest node receives some data from the dealer, even in the asynchronous setting. If this data turns out to be invalid, it can be used as evidence to implicate the dealer. Once the dealer is determined to be faulty, we enter a share recovery phase, which ensures every correct party receives their share. The share recovery phase can be very efficient too, since we do not ensure the confidentiality of a malicious dealer's shares.

c) Avoiding Trusted Setup: Our ACSS protocol can be instantiated with any polynomial commitment scheme, but we focus on two: first, a state of the art scheme from Tomescu et al. [57], and second, our own scheme which avoids trusted setup. Tomescu et al.'s scalable VSS scheme is built around a polynomial commitment scheme based on authenticated multipoint evaluation trees (AMTs) that achieves quasilinear overhead; i.e., for AMT, the resources incurred at each server—communication and computation—do not increase significantly as the system setup grows. However, a limitation of this scheme is that it relies on public parameters that must be generated through a sampling process, where intermediate values during this process must be securely erased or else the resulting protocol is insecure. Such "trusted setups" are a significant obstacle to deployment of a distributed system, requiring difficult-to-coordinate setup ceremonies [19], [18] that need to be repeated if parameters are changed.

Our construction, hbPolyCommit, is based on Bulletproofs [21] and achieves similar asymptotic and practical performance as AMT while requiring only uniform reference strings and the discrete log assumption. Beyond VSS, polynomial commitments are also widely used in applications such as SNARKs [29] and may be of independent interest.

To complement our constructions and asymptotic analysis, we empirically evaluate both hbACSS and hbPolyCommit. We show that hbPolyCommit is comparable in performance with AMT, eliminating the need for a trusted setup in practice. When used to instantiate hbACSS, we find that we can robustly generate input masks at a rate of ∼39 input masks per CPU-second when $N = 31$ and ∼9 per second at $N = 127$.

d) Contributions: To summarize our contributions: we design and implement a set of ACSS schemes which simultaneously achieve better asymptotic bounds than previous work as well as demonstrate the growing practicality of asynchronous, optimally-resilient MPC. We instantiate our ACSS schemes with a batch-optimized polynomial commitment scheme that achieves similar performance to state-of-the-art work while negating the need for a trusted setup. Lastly we use our new constructions to demonstrate the robust computation of a useful MPC primitive.

II. Preliminaries

A. Threat Model

We assume the standard asynchronous fully-connected network of N parties $\{\mathcal{P}_1, ..., \mathcal{P}_N\}$ [22]. A special party D works as a dealer, which can either be one of N parties or the $N+1$th party. The indices for N parties are chosen from $\mathbb{F}_p$. Without loss of generality, we assume these to be $1, \ldots, N$.

Every pair of parties is connected by an authenticated communication channel. The adversary can corrupt and coordinate the actions of up to t out of N parties (we assume the optimal asynchronous Byzantine fault tolerance of $N = 3t + 1$). If the dealer D is the $N + 1$th party, the adversary may additionally compromise the dealer. The adversary is assumed to be static, and chooses the parties it wishes to corrupt at the beginning of the protocol execution. A party is said to be correct if the adversary has not corrupted it. The adversary controls the network and may delay messages between any two correct parties. However, it cannot modify messages, and it also has to eventually deliver messages from correct parties.

Lastly, we assume the adversary is bounded computationally by security parameter κ such that the adversarial advantage of breaking the security of the protocol is negligible in κ.

B. Asynchronous Complete Secret Sharing

Here we give our security definition for Asynchronous Complete Secret Sharing (ACSS). Compared to Asynchronous Verifiable Secret Sharing (AVSS), which only guarantees that parties can reconstruct the secret s, ACSS also guarantees that the parties can reconstruct the entire secret sharing polynomial ϕ associated with it, which is necessary when using this primitive for MPC applications.

Definition 1. *(Asynchronous Complete Secret Sharing—ACSS [52]) In an ACSS protocol, the dealer D receives input $s \in \mathbb{F}_p$, and each party P_i receives a share $\phi(i)$ for some degree-t polynomial $\phi : \mathbb{F}_p \to \mathbb{F}_p$.*

The protocol must satisfy the following properties

- **Correctness**: If the dealer D is correct, then all correct parties eventually output a share $\phi(i)$ where ϕ is a random polynomial with $\phi(0) = s$.
- **Secrecy**: If the dealer D is correct, then a computationally bounded adversary learns no information about ϕ except for the shares of corrupted parties, except with negligible probability.
- **Agreement**: If *any* correct party receives output, then there exists a unique degree-t polynomial ϕ' such that each correct party $\mathcal{P}_i$ eventually outputs $\phi'(i)$.

For simplicity, this definition is specific to Shamir sharing, though a more generic definition based on linear secret sharing is possible. Our agreement property incorporates the

completeness property of [52], enforcing that all honest parties must hold a valid share at the end of the sharing phase.

C. Polynomial Commitments

One of the main building blocks we use are Polynomial Commitment (PolyCommit) schemes. This primitive enables a committer to commit to a polynomial. The commitment can be checked by a verifier to assert correctness of claimed evaluation points on the committed polynomial. We formally define this primitive as follows:

Definition 2. *(Polynomial Commitment—PolyCommit [42]) Let $(\mathbb{F}_p)_\kappa$ be a family of finite fields indexed by a security parameter κ (we typically omit κ and write $\mathbb{F}_p$). A PolyCommit scheme for $\mathbb{F}_p$ consists of the following five algorithms:*

- $\mathsf{Setup}(1^\kappa, t) \to \mathsf{SP}$: generates system parameters SP to commit to a polynomial over $\mathbb{F}_p$ of degree bound t. Setup is run by a trusted or distributed authority. SP can also be standardized for repeated use.
- $\mathsf{PolyCommit}(\mathsf{SP}, \phi(\cdot)) \to (C, \mathsf{aux})$: outputs a commitment C to a polynomial $\phi(\cdot)$ for system parameters SP and some associated scheme-specific decommitment information aux.
- $\mathsf{VerifyPoly}(\mathsf{SP}, C, \phi(\cdot), \mathsf{aux}) \to$ *bool*: verifies that C is a commitment to $\phi(\cdot)$, created with decommitment information aux. The algorithm *accepts* if verification succeeds, and *rejects* otherwise.
- $\mathsf{ProveEval}(\mathsf{SP}, \phi(\cdot), i, \mathsf{aux}) \to (i, \phi(i), \pi_i)$: outputs a index i, evaluation point $\phi(i)$, and a proof π_i for the evaluation $\phi(i)$ of $\phi(\cdot)$ at the index i.
- $\mathsf{VerifyEval}(\mathsf{SP}, C, i, \phi(i), \pi_i) \to$ *bool*: verifies that $\phi(i)$ is indeed the evaluation at index i of the polynomial committed in C. If verification succeeds, the algorithm *accepts*, and otherwise *rejects*.

We use the functions $\mathsf{BatchProveEval}$ and $\mathsf{BatchVerifyEval}$ to allow participants to prove (resp. verify) many evaluations at once. If specific batch functions are not defined, we use many invocations of $\mathsf{ProveEval}$ and $\mathsf{VerifyEval}$ instead.

A valid PolyCommit scheme must satisfy the following:

- **Correctness**: If $C, \mathsf{aux} \leftarrow \mathsf{Commit}(\mathsf{SP}, \phi(\cdot))$ and $\pi_i, \mathsf{aux}_i \leftarrow \mathsf{ProveEval}(\mathsf{SP}, \phi(\cdot), i, \mathsf{aux})$, then the correct evaluation of $\phi(i)$ is successfully verified by $\mathsf{VerifyEval}(\mathsf{SP}, C, i, \phi(i), \pi_i, \mathsf{aux}_i)$.
- **Polynomial Binding**: If $C, \mathsf{aux} \leftarrow \mathsf{Commit}(\mathsf{SP}, \phi(\cdot))$, then except with negligible probability, an adversary cannot create a polynomial $\phi'(\cdot)$ such that $\mathsf{VerifyPoly}(\mathsf{SP}, C, \phi(\cdot)', \mathsf{aux}) = 1$ if $\phi(\cdot) \neq \phi'(\cdot)$.
- **Strong Evaluation Binding**: Any commitment and evaluation proofs generated by an adversary must be consistent with some degree-t polynomial. Formally,

$$\Pr\left[\begin{array}{l}\mathsf{SP} \leftarrow \mathsf{Setup}(1^\kappa, t)\\ (C, \{x_i, y_i, \pi_i\}_{i\in[1..\ell]}) \leftarrow \mathcal{A}(\mathsf{SP})\\ \forall i \in [1..\ell].\\ \mathsf{VerifyEval}(\mathsf{SP}, C, x_i, y_i, \pi_i) = 1 \wedge\\ \nexists \text{ deg-}t\ \phi(\cdot)\ s.t.\\ \forall i \in [1..\ell].\ y_i = \phi(x_i)\end{array}\right] \leq \mathsf{negl}(\kappa)$$

That is, given

$$C, \mathsf{aux} \leftarrow \mathsf{Commit}(\mathsf{SP}, \phi(\cdot)), \text{ and}$$
$$\pi_i, \mathsf{aux}_i \leftarrow \mathsf{ProveEval}(\mathsf{SP}, \phi(\cdot), i, \mathsf{aux}),$$

except with negligible probability, an adversary cannot create an evaluation $\phi(j)$, proof π_j, and decommitment information aux_j such that $\mathsf{VerifyEval}(\mathsf{SP}, C, i, \phi(j), \pi_j, \mathsf{aux}_j) = 1$ if $i \neq j$.

- **Zero-Knowledge**: Informally, the commitment and evaluation proofs should not reveal any information about the polynomial beyond what is implied by public information. Formally, there must exist a simulator $(\mathsf{Sim}_1, \mathsf{Sim}_2)$, such that for all adversaries $\mathcal{A}$, the following two distributions are (computationally or information-theoretically) similar—we refer to the information-theoretic case as *perfect* zero-knowledge. We use this stronger definition in place of the hiding definition introduced by Kate et al. [42].

Real World:

$$\left\{\begin{array}{l}\mathsf{SP} \leftarrow \mathsf{Setup}_1(1^\kappa, t)\\ (\phi(\cdot), \{x_i\}_{i\in[1..\ell]}) \leftarrow \mathcal{A}(\mathsf{SP})\\ (C, \mathsf{aux}) \leftarrow \mathsf{PolyCommit}(\mathsf{SP}, \phi(\cdot))\\ \forall i \in [1..\ell].\ \pi_i \leftarrow \mathsf{ProveEval}(\mathsf{SP}, \phi(\cdot), x_i, \mathsf{aux})\\ : (C, \{x_i, y_i, \pi_i\}_{i\in[1..\ell]})\end{array}\right\}$$

Ideal World:

$$\left\{\begin{array}{l}(\mathsf{SP}, \mathsf{st}) \leftarrow \mathsf{Sim}_1(1^\kappa, t)\\ (\phi(\cdot), \{x_i\}_{i\in[1..\ell]}) \leftarrow \mathcal{A}(\mathsf{SP})\\ (C, \{\pi_i\}) \leftarrow \mathsf{Sim}_2(\mathsf{st}, \{x_i, \phi(x_i)\}_{i\in[1..\ell]})\\ : (C, \{x_i, y_i, \pi_i\}_{i\in[1..\ell]})\end{array}\right\}$$

D. Arguments of Knowledge

We make use of zero knowledge proofs in the contruction of our polynomial commitment scheme. In particular, we start with public-coin interactive arguments of knowledge, where the verifier $\mathcal{V}$ chooses its messages uniformly at random and independent of the prior messages sent by $\mathcal{P}$, and apply Fiat-Shamir as a last step in our construction.

Notation. Let $\mathbb{G}$ denote a cyclic group of prime order p. We use bold font for vectors, i.e. $\mathbf{a} \in \mathbb{Z}_p^n$ is a vector with elements $a_1, a_2, \ldots, a_n \in \mathbb{Z}_p$. For a scalar $c \in \mathbb{Z}_p$ and a vector $\mathbf{a} \in \mathbb{Z}_p^n$, we compute $\mathbf{b} := c \cdot \mathbf{a} \in \mathbb{Z}_p^n$ by scaling each element in $\mathbf{a}$ by c, i.e. $b_i := c \cdot a_i$ for $i \in [n]$. Let $\langle \mathbf{a}, \mathbf{b} \rangle := \sum_{i=1}^n a_i \cdot b_i$ denote the inner product between two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{Z}^n$. Let $\mathbf{ab} = \mathbf{a} \cdot \mathbf{b} := (a_1 \cdot b_1, a_2 \cdot b_2, \ldots, a_n \cdot b_n) \in \mathbb{Z}_p^n$ be the entry-wise multiplication of two vectors. For a vector $\mathbf{g} = (g_1, g_2, \ldots, g_n) \in \mathbb{G}^n$ and $\mathbf{a} \in \mathbb{Z}_p^n$ we write $\mathbf{g}^\mathbf{a} = \prod_{i=1}^n {g_i}^{a_i} \in \mathbb{G}$. For $0 \leq i \leq j \leq n$, we use Python notation to denote slices of vectors as follows:

$$\mathbf{a}_{[i:j]} = (a_i, a_{i+1}, \ldots, a_{j-1}) \in \mathbb{Z}_p^{j-i}$$

where $\mathbf{a}$ is an n-length vector and indexing begins with zero. For some integer i and vector $\mathbf{a} \in \mathbb{Z}_p^n$, $\mathbf{a}_{[i]}$ refers to the $(i \bmod n)$-th entry in $\mathbf{a}$. We often use $\mathbf{a}_{[-1]}$ to refer to the last item in the array. We define $\mathbf{a}^z := (za_1, za_2, \ldots, za_n)$ for $\mathbf{a} \in \mathbb{Z}_p^n$ and $z \in \mathbb{Z}_p$. Denote the concatenation of two vectors $\mathbf{a} \in \mathbb{Z}_p^i$, $\mathbf{b} \in \mathbb{Z}_p^j$ as $\mathbf{a} \mid\mid \mathbf{b} \in \mathbb{Z}_p^{i+j}$.

Let $\{R[\mathsf{crs}]\}_\mathsf{crs}$ be a family of polynomial-time decidable relations indexed by a string crs. We call w a witness for a statement stmt with respect to the relation $R[\mathsf{crs}]$ if $(\mathsf{stmt}, w) \in R[\mathsf{crs}]$. We often use $R := R[\mathsf{crs}]$ as shorthand. We write $\{\text{Public Input}; \text{Witness} : \text{Relation}\}$ to denote the relation Relation using the specified Public Input and Witness.

For an interactive proof system $\langle \mathsf{Gen}(1^\kappa), \mathcal{P}, \mathcal{V} \rangle$, the following properties must hold:

Definition 3. *Perfect Completeness. For every* $\kappa \in \mathbb{N}$*, every* crs *in the support of* $\mathsf{Gen}(1^\kappa)$*, and every* $(\mathsf{stmt}, w) \in R[\mathsf{crs}]$*,*

$$\Pr[\langle \mathcal{P}(\mathsf{crs}, \mathsf{stmt}, w), \mathcal{V}(\mathsf{crs}, \mathsf{stmt})\rangle = 1] = 1.$$

Definition 4. *Knowledge Soundness. For every deterministic, polynomial-time prover* $\mathcal{P}^*$*, there exists an expected polynomial-time extractor* $\mathcal{E}_{P^*}$*, and negligible function* $\mathsf{negl}(\cdot)$ *such that for every* stmt *and every* z*, and every* $\kappa \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{l} \mathsf{crs} \leftarrow \mathsf{Gen}(1^\kappa), \\ tr \leftarrow \langle \mathcal{P}^*(\mathsf{crs}, \mathsf{stmt}, z), \mathcal{V}(\mathsf{crs}, \mathsf{stmt})\rangle, \\ w \leftarrow \mathcal{E}_{\mathcal{P}^*}(\mathsf{crs}, \mathsf{stmt}, z) : \\ \text{if } tr \text{ is accepting then } (\mathsf{stmt}, w) \in R \end{array}\right] \geq 1 - \mathsf{negl}(\kappa)$$

E. Asynchronous Verifiable Information Dispersal

Our protocol relies on an information dispersal protocol [23] as specified below. Our definition is for a batch such that M messages $v_1, ..., v_M$ are dispersed at once and can be individually retrieved.

Definition 5. *(Asynchronous Verifiable Information Dispersal—AVID [23]) A* $(t+1, N)$ *AVID scheme* AVID *for* M *values is a pair of protocols* $(\mathsf{Disperse}, \mathsf{Retrieve})$ *which satisfy the following functionality:*

- Disperse*: A message is split into* N *blocks, each one being stored by one of the* N *protocol participants.*
- Retrieve*: Message blocks are requested from the other participants until there is sufficient information to fully reconstruct the original message.*

The following properties must be satisfied with high probability:

- **Termination:** If the dealer D is correct and initiates $\mathsf{Disperse}(v_1, ..., v_M)$, then every correct party eventually completes Disperse.
- **Agreement:** If any correct party completes Disperse, all correct parties eventually complete Disperse.
- **Availability:** If $t+1$ correct parties have completed Disperse, and some correct party initiates $\mathsf{Retrieve}(i)$, then the party eventually reconstructs a message v'_i.
- **Correctness:** After $t+1$ correct parties have completed Disperse, then for each index $i \in [M]$ there is a value v_i such that if a correct party receives v'_i from $\mathsf{Retrieve}(i)$, then $v'_i = v_i$. Furthermore, if the dealer is correct, then v_i is the value input by the dealer.

In particular, we use the $\mathsf{AVID-H}$ protocol from [23], in which the total communication complexity is only $O(|v|)$ in the Disperse phase for a sufficiently large batch $v \gg N \log N$. That is, it achieves only constant communication overhead — this property is essential to reaching our asymptotic goals.

F. Reliable Broadcast

Reliable broadcast [20] allows a dealer D to broadcast a message v to every party. Regardless of whether the dealer is correct, if any party receives some output v', then every party eventually receives v'. Reliable broadcast is a special case of information dispersal, where each party simply begins Retrieve immediately after Disperse completes. In fact, all efficient protocols we know of, such as [23] or [39], are built from an AVID protocol. We therefore skip the definition but use the ReliableBroadcast syntax in our protocol description as shorthand for Disperse followed by all parties immediately beginning Retrieve.

G. Public-Key Encryption

We use a semantically secure public-key encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, such that $\mathsf{Enc}_{\mathsf{PK}}(m)$ produces a ciphertext encrypted under public key PK, while $\mathsf{Dec}_{\mathsf{SK}}(c)$ decrypts the message using secret key SK. We assume a PKI, such that each party $\mathcal{P}_i$ already knows SK_i. We also assume that each public key is a function of the secret key, written $\mathsf{PK} = g^{\mathsf{SK}}$.

III. Related Work: The Many Settings for Verifiable Secret Sharing

Verifiable secret sharing (VSS) has been studied in many different security models. To help place our work in context, we review the related work in the most relevant settings.

A. Completeness (Share Recovery)

An important design goal for our protocol is *Completeness* [52], which essentially guarantees that every honest party receives their share of $[\![v]\!]$. While necessary for our target application of MPC, this property is not needed for all use cases of VSS, such as the backup storage and key management of Unbound [46]. In MPC however, share unavailability can cause the protocol to be aborted.

Although we use the terminology from [31], the completeness property appears earlier in [14] where it is known as *Ultimate Secret Sharing*. In this work, an AVSS scheme is invoked n times to essentially secret-share every individual share, so that all honest parties can reconstruct their share if needed. Recently, [9] give a protocol for this setting called VSS with Share Recovery (VSS-R).

B. Adversarial Assumptions

a) Unconditional vs Computational Security: The most aggressive adversarial assumption for VSS is a computationally-unbounded Byzantine adversary, or *information-theoretic security*. There are a variety of such VSS protocols for both the synchronous [13] and asynchronous [25] network settings. Information-theoretic VSS can tolerate a maximum of $t < \lfloor N/3 \rfloor$ Byzantine faults (though such an asynchronous protocol must have a non-zero probability of failing to terminate[1]). Information-theoretic protocols may either be *perfectly secure* [53], [32] (i.e. protocols with zero failure probability) or statistically secure [52] (with some failure probability). The best optimally-resilient, statistically secure ACSS protocol we know of in the information-theoretic setting [31] has a bandwidth complexity of $O(N^3)$.

All of the unconditionally secure VSS protocols require additional communication rounds due to the need for interactive proofs [55]. The use of cryptography based on computational assumptions can lower communication costs both in round complexity and total bandwidth [9], [48], [7], [57]. In fact, our use of such cryptography allows us to improve our amortized network bandwidth by a factor of at least $O(N)$ over all information theoretic ACSS protocols we know of.

C. Network Models

There are various choices of network models which significantly affect the achievable fault tolerance and performance of VSS protocols. The most common network assumption is synchrony (i.e. a strictly bounded network propagation time), which is useful since nodes can be timed out, allowing for a fault tolerance of $t < \lfloor N/2 \rfloor$. The primary drawback of synchronous networks is that if the network is briefly unresponsive, it can lower the resulting fault tolerance. A related drawback of synchronous protocols is the need to tune the timeout parameter. If this parameter is set too small, it risks ejecting honest parties. If a timeout parameter is too large, performance may suffer. Asynchronous protocols avoid this dilemma.

The protocol from [9] works in a partially synchronous network setting. Their protocol is safe and live even in an asynchronous network, but in an asynchronous network it may fall back to a less-efficient backup mode.

D. Batching

Most VSS protocols are written to share a single secret, and this is sufficient for many applications such as distributed key generation or small queries to an MPC-based service. Batched secret sharing has received comparatively less attention, much of which has been in Packed Secret Sharing (PSS).

a) Packed Secret Sharing: In PSS, a single polynomial has multiple evaluation points that correspond to secrets. To achieve this, the degree of the sharing polynomial must be $t + b - 1$, where t is the fault tolerance threshold and b is the number of secrets shared.

PSS inherently requires weakening the fault tolerance threshold as the polynomials must be of a larger degree than normal, but at the same time, allows for more secrets to be packed as the number of players increases. For example, using a fault tolerance of $N = 4t + 1$ in the asynchronous setting would allow each polynomial to encode 3 secrets when $N = 9$, but 26 secrets when $N = 101$. Despite this, some recent VSS schemes leverage PSS [36], [27], [8]. Notably, Patra et al. [54] use packing to shave off an $O(N)$ factor in their AVSS scheme in the $N = 4t + 1$ setting.

The goal of our work is to optimize the performance of AVSS at the optimal Byzantine fault tolerance setting. However, we do feel that lessening the fault tolerance to increase overall throughput is worth exploring in future work.

b) Dual-Threshold VSS: The work of Kokoris-Kogias et al [44] introduced the first *dual-threshold* AVSS scheme, meaning that the reconstruction threshold can be made larger than the privacy threshold. The later work of AlHaddad [4] improves upon this by reducing the bandwidth requirements. Both of these schemes accomplish their dual-threshold property through the use of bivariate polynomials, which consequently result in at least $O(N^2)$ network bandwidth as N univariate polynomials need to be sent to N parties. The later work also proposes a batch optimization in which a large symmetrically-encrypted message is efficiently broadcasted and the key is verifiably secret shared. This can trivially batch-amortize any computational VSS protcol to be $O(N)$ but the shares are now non-homomorphic fragments of ciphertexts which lack properties that make them useful for applications such as MPC.

c) Batched Single-Share Polynomials: Batched secret sharing through the use of many polynomials at once has been explored in previous research [54], albeit with a differing end goal. While this work explored batched VSS to improve the throughput of existing protocols, our end goal is to use batching to bring down the overhead of asynchronous VSS to incentivize the development and deployment of network-robust multiparty protocols.

E. Polynomial Commitments in VSS

The choice of a polynomial commitment scheme used to instantiate a VSS protocol has large implications for the overall computational work and bandwidth. While polynomial commitment schemes have received recent research interest due to their usefulness in circuit satisfiability arguments, schemes such as [16], [17] will not be considered in this analysis, as they are only intended to be used to prove a polynomial evaluation at a single point. Instead, we will focus on schemes derived from the original PolyCommit scheme of Kate, Zaverucha, and Goldberg (KZG)[42] and other schemes which do not require a structured CRS.

a) KZG-like Schemes: KZG polynomial commitments are a common first choice due to being both constant-sized and additively homomorphic. While these properties translate well to protocols that attempt to achieve new asymptotic communication bounds, they require $O(N)$ elliptic curve point multiplications per evaluation proof, which can slow down actual VSS implementations (both we and the authors of [9] noticed that $O(t)$-sized polynomial commitments were faster in practice because of this).

Fortunately, the recent work of [57] addresses this issue through the development of *authenticated multipoint evaluation trees* (AMTs). More specifically, proof computation cost is lowered considerably at the cost of $O(\log(t))$-sized evaluation proofs, with the trade-off of the proof size and verification times becoming logarithmic. We consider this to be the current state of the art for KZG-style PolyCommit schemes.

b) Schemes with an Unstructured CRS: Furthermore, KZG-style PolyCommits have other nonideal properties such as nonstandard hardness assumptions and reliance on trusted setup. In fact, the authors of CHURP [48] thought that these problems were significant enough to design their protocol to have a backup operational mode if it detected that KZG assumptions were violated. Further, while a trusted setup is not an unsolved problem [19], [18] it does present a logistical hurdle to deployment and may need to be redone on protocol redesign or possibly even an increase in the number of parties.

In contrast, an unstructured CRS does not require trusted setup and can instead consist of nothing-up-my-sleeve numbers. Pedersen's VSS scheme [55] was the first VSS to use non-interactive evaluation proofs and also happens to feature the first polynomial commitment scheme with an unstructured CRS. While this work predates KZG polynomial commitments, the cryptography involved can be fit into the PolyCommit interface to create a scheme with lightweight operations at the cost of $O(N)$ size and verification, a trade-off worth making even in recent works [9]. Our PolyCommit scheme, hbPolyCommit, aims to be similarly useful, but with improved performance characteristics. We compare hbPolyCommit with similar work in Table I.

	Prover Comp	Verifier Comp	Proof Size
PolyCommitPed [42]	$O(N)$	$O(1)$	$O(1)$
PolyCommitHB	$O(N)$	$O(N)$	$O(\log N)$
PolyCommitHB-Batch	$O(N)$	$O(\log N)$	$O(\log N)$
AMT PolyCommit [57]	$O(\log N)$	$O(\log N)$	$O(\log N)$

TABLE I: Amortized Asymptotic Behaviors of PolyCommit Schemes

IV. Efficient Polynomial Commitments without Trusted Setup

Our ACSS construction uses polynomial commitments to abstract away the cryptographic components of share validation. Most polynomial commitment schemes, dating back to [42] and including state-of-the-art AMTs [57], rely on a trusted setup. Our goal in this section is to design a polynomial commitment scheme that can achieve similar performance to the state-of-the-art, but without requiring a trusted setup ceremony. The resulting performance of hbACSS largely relies on the performance of this primitive.

A. Bulletproofs for Polynomial Evaluation

We follow an approach by [58] which is to build polynomial commitments based on Bulletproofs [21], a recent innovation for proofs involving inner-products that does not require a trusted setup. To summarize with Camenisch-Stadler notation [24], a Bulletproof can provide a computationally sound argument for the following relation:

$$\{(\mathbf{a} \in \mathbb{Z}_p^q) : A = \mathbf{g}^{\mathbf{a}} \wedge v = \langle \mathbf{a}, \mathbf{y} \rangle\} \quad (1)$$

where $\mathbf{g} \in \mathbb{G}^q$ are public parameters, $A \in \mathbb{G}$ is a commitment to a polynomial given by coefficients $\mathbf{a}$, v is the purported evaluation result, and $\mathbf{y} = 1, i, ..., i^{q-1}$ defines the evaluation point. Put another way, it proves that $v = \phi(i)$ where the committed polynomial is $\phi(X) = \mathbf{a}_0 + \mathbf{a}_1 X + ...\mathbf{a}_{q-1} X^{q-1}$.

B. Batching verification of multiple evaluations

Directly using the Bulletproofs inner-product argument would provide a logarithmic proof size, but without batch verification, the verification cost would be too high for our purposes. We can describe the batching needed in our ACSS setting as follows,

$$\{[P \leftrightarrow V_i]\ (\mathbf{a} \in \mathbb{Z}_p^{B \times q}) : A = \mathbf{g}^{\mathbf{a}_{j,:}} \wedge v = \langle \mathbf{a}_{\mathbf{j},:}, \mathbf{y}_{:,\mathbf{i}} \rangle\}_{i,j} \quad (2)$$

where $i \in [1..N]$ ranges over the verifiers in the protocol, and $j \in [1..B]$ ranges over all the secrets to be shared. That is, we are sharing B polynomials $\{\mathbf{a}_j\}$, and each verifier V_i is responsible for checking all the polynomials at a given evaluation point $\mathbf{y}_i$.

In Figure 7 in the Appendix we give the base protocol for hbPolyCommit. For readers familiar with Bulletproofs, most of the protocol is a straightforward adaptation of the inner-product argument of that work. The protocol is interactive, though Fiat-Shamir can be applied at a later stage. Similarly the base proof is not zero-knowledge, although this can be fixed in a layered way by rerandomizing the polynomial before proving it.

In the inner-product argument protocol, the group element vector $\mathbf{g}$ is folded in half at each recursive step in response to a challenge. This operation requires $2q$ group exponentiations overall, and different challenges result in new folding computations that need to be performed for every proof. However, if we were able to *reuse* the set of challenges for a batch of proofs, then the vector folding would only need to be calculated once for the whole batch.

hbPolyCommit-Core (Batch Inner-Product Proof)

Let $\mathbf{a}$ be a $B \times q$ matrix, and $\mathbf{y}$ a $q \times N$ matrix
Each verifier $\mathcal{V}_i$ knows the i'th column, $\mathbf{y}_{:,i}$.

`batch_inner_product_proof`(stmt):

1. *Setup.* Run $\mathbb{G} \leftarrow \mathcal{G}(1^\kappa)$ and let $\mathbf{g} := (g_0, \ldots, g_{q-1})$, and $\mathsf{crs} := (\mathbb{G}, \mathbf{g}, h)$.
2. *Input.* Both $\mathcal{P}$ and $\mathcal{V}_i$ know the statement $\mathsf{stmt}_i := (\mathbf{A}_{:,i}, \mathbf{y}_{:,i}, \mathbf{v}_{:,i})$ and $\mathcal{P}$ knows a witness $\mathbf{a}$ such that $\mathbf{v} := \mathbf{a} \cdot \mathbf{y}$ and $A_{j,i} := \mathbf{g}^{\mathbf{a}_{\mathbf{j},:}} \cdot h^{\mathbf{v}_{j,i}}$.
3. let $z_i = \mathcal{H}(\mathsf{stmt}_i)$
4. $\mathcal{P}$ and $\mathcal{V}_i$ compute $A'_{ji} := A_{ji} \cdot h^{z_i \cdot v_{ji}}$.
   Return `batch_reduce_proof`$(\mathsf{crs}, \mathbf{A}', \mathbf{y}, q; \mathbf{a})$.

`batch_reduce_proof`$(\mathsf{crs}, \mathbf{A}, \mathbf{y}, q; \mathbf{a})$:

1. if $q = 1$:
1.1. $\mathcal{P}$ sends $\mathbf{a}$ to $\mathcal{V}_i$.
1.2. $\mathcal{V}_i$ returns the result of $\bigwedge_{j=1}^{B}(A_{ji} \stackrel{?}{=} g^{a_{j,:}} \cdot h^{a_{j,:} \cdot y_{:,i}})$.
2. if q is odd:
2.1. $\mathcal{P}$ sends $\mathbf{na} := -\mathbf{a}_{:,q}$ to $\mathcal{V}_i$.
2.2. $\mathcal{P}$ sets $\mathbf{a} := \mathbf{a}_{:,1:q}$.
2.3. $\mathcal{P}$ and $\mathcal{V}_i$ update $\mathbf{A}$, $\mathbf{y}$, $\mathbf{g}$, and q as follows:

$$A_{ji} := A_{ji} \cdot g_q^{\mathbf{na}_{ji}} \cdot h^{(\mathbf{na}_{ji} \cdot \mathbf{y}_{qi})}$$
$$\mathbf{y} := \mathbf{y}_{1:q,:}, \quad \mathbf{g} := \mathbf{g}_{1:q}, \quad q := q - 1$$

3. Let $q' = q/2$ and
   $\mathbf{a}_L = \mathbf{a}_{:,1:q'}$ and $\mathbf{a}_R = \mathbf{a}_{:,q':q}$ be $B \times q'$ matrices
   and $\mathbf{y}_L = \mathbf{y}_{1:q',:}$ and $\mathbf{y}_R = \mathbf{y}_{q':q,:}$ be $q' \times N$ matrices
4. $\mathcal{P}$ computes:

$$\mathbf{c}_L := \mathbf{a}_L \cdot \mathbf{y}_R, \quad \mathbf{c}_R := \mathbf{a}_R \cdot \mathbf{y}_L$$
$$\mathbf{L}_{ji} := \mathbf{g}_L{}^{\mathbf{a}_{Lj,:}} \cdot h^{\mathbf{c}_{Lji}}, \quad \mathbf{R}_{ji} := \mathbf{g}_R{}^{\mathbf{a}_{Rj,:}} \cdot h^{\mathbf{c}_{Rji}}$$

   and sends $\{\mathbf{L}_{:,i}, \mathbf{R}_{:,i}\}$ to $\mathcal{V}_i$.
5. $\mathcal{P}$ builds a Merkle Tree over all transcript sets where: $\mathit{leaf}_i = \mathcal{H}(\mathbf{g}, q, h, \mathbf{y}_{:,i}, A_{:,i}, L_{:,i}, R_{:,i}, na_{:,i})$
6. $\mathcal{P}$ sends $\mathcal{V}_i$ $\{z := \mathit{roothash}, b_i := \mathsf{MerkleBranch}(i)\}$.
7. $\mathcal{V}_i$ calculates leaf_i and returns False if: $!\mathsf{MerkleVerify}(\mathit{leaf}_i, z, b_i)$
8. $\mathcal{P}$ and $\mathcal{V}_i$ both compute:

$$A'_{ji} := L_{ji}^{z^2} \cdot A_{ji} \cdot R_{ji}^{z^{-2}}, \quad \mathbf{g}' := \mathbf{g}_L^{z^{-1}} \cdot \mathbf{g}_R^{z}$$
$$\mathbf{y}'_{:,i} := z^{-1} \cdot \mathbf{y}_{L:,i} + z^1 \cdot \mathbf{y}_{R:,i}$$

9. $\mathcal{P}$ computes $\mathbf{a}'_{j,:} := z^1 \cdot \mathbf{a}_{Lj,:} + z^{-1} \cdot \mathbf{a}_{Rj,:}$.
10. Return `batch_reduce_proof`$((\mathbb{G}, \mathbf{g}', h), A', \mathbf{y}', q'; \mathbf{a}')$.

Fig. 1: A protocol for proving many inner-product evaluations to many verifiers where the $\mathbf{y}$ matrix is public.

The Fiat-Shamir heuristic by itself can accommodate some degree of parallel challenge reuse. Consider a prover who wishes to make several simultaneous evaluation proofs to a verifier. If many proofs are made in parallel, then the transcript for each could be included in the hash function that maps to a challenge. This alone reduces the verifier's amortized total computation from $O(q)$ to $O(\log(q))$.

We take this idea a step further and use the same set of

challenges for all proofs for all verifiers. Since we cannot send every transcript to every verifier, when a challenge is required, we build a Merkle tree where each leaf contains all of the transcripts of a given verifier and the roothash serves as the challenge. Then we send each verifier a Merkle branch, allowing the verifier to reconstruct the roothash and verify that it fully incorporates all of the verifier's transcripts.

We show this modification in Figure 1, where we express a prover who is providing B different length-q inner product arguments to N different verifiers as a series of matrix operations. As this protocol uses extensive matrix slicing, we refer to $\mathbf{M}_{i,:}$ as the $i'th$ row of a matrix $\mathbf{M}$, $\mathbf{M}_{:,j}$ as the j'th column, $\mathbf{M}_{1:k,:}$ as rows 1 to $k-1$ and so on, while M_{ij} denotes a single element.

Using Merkle trees introduces communication and computation overhead which are amortized away given a linear batch size (as they are only performed once, regardless of the batch size). This technique reduces the prover's group exponentiation work from $O(q)$ to $O(\log(q))$ per proof. While this does not reduce the prover's overall computation complexity (due to the dot products in part 3 of `reduce_proof`) it does offer a large practical speed-up due to the remaining $O(t)$ operations being very fast.

a) Making it zero knowledge: So far the scheme we have presented is not zero knowledge, and in particular the statement A itself reveals information about the witness $\mathbf{a}$. In order to instantiate the PolyCommit, we need to blind the statement before `batch_reduce_proof` is called. This is achieved by extending the wrapper function (`batch_inner_product_proof`) with the following: the prover must convince the verifier that, in addition proving that $\mathbf{a}$ is a valid opening of A, it knows some u such that $u = \langle \mathbf{a}, \mathbf{y} \rangle$ and $U = g^u$. This enables implicit verification of the original statement, as it can be instead verified in the exponent. We give more detail on this technique in Appendix A.

V. The hbACSS Protocols

A. Protocol description

We present our main construction in three incremental variants hbACSS0, hbACSS1, hbACSS2. Our simplest protocol hbACSS0 can be instantiated with any polynomial commitment scheme that supports at least $t+1$ evaluations. This starting point is asymptotically similar to VSS-R [9] in that it achieves linear overhead in the optimistic case, but quadratic overhead in the worst case. The next protocol hbACSS1 achieves better batching in the worst case as well, but relies on a homomorphic polynomial commitment scheme, which precludes (among others) our polynomial commitment with no trusted setup in (see Appendix C1). Finally, hbACSS2 achieves the same asymptotic costs as hbACSS1 but restores the flexibility to use any polynomial commitment.

All hbACSS protocols closely follow the same main steps:

1) Dealer's phase: the dealer computes each of the B secrets from polynomials and broadcasts the corresponding polynomial commitments. She then encrypts each party's shares and evaluation proofs using their public keys and verifiably sends them via an AVID scheme.
2) Share validation: each party retrieves their encrypted payload and attempts to decrypt and validate their shares against the polynomial commitments. If sufficiently many parties successfully receive valid shares, then the shares are output.
3) Implicating a faulty dealer: if any party finds that the shares they receive are invalid or fail to decrypt, they reveal their secret key, enabling the other parties to confirm that the dealer was faulty after retrieving that party's payload from the dispersal scheme.
4) Share recovery: once the dealer is implicated as faulty, the parties who did receive valid shares distribute them to enable the remaining parties to reconstruct their shares.

We now explain hbACSS0 in more detail, following along with the protocol pseudocode given in Algorithm 1. Security analysis and the other variants follow in this section.

1) Sharing and Committing: The protocol shares a batch of B inputs at a time, $\{s_1, ..., s_B\}$. The dealer creates a degree-t Shamir sharing $\phi_k(\cdot)$ for each input such that $\phi_k(0) = s_k$, and each party $\mathcal{P}_i$'s share of s_k is $\phi_k(i)$.

The dealer then uses the PolyCommit procedure to create a commitment C_k to each polynomial $\phi_k(\cdot)$. The commitments are then broadcasted, ensuring all the parties can validate their shares against the same set of commitments.

Next, for each party $\mathcal{P}_i$, the dealer creates an encrypted payload z_i, consisting of the shares $\{\phi_k(i)\}_{k\in[1,B]}$ and the polynomial evaluation proof π_i, encrypted under $\mathcal{P}_i$'s public key PK_i. The dealer then Disperses these encrypted payloads. With the broadcast and dispersal complete, the dealer's role in the protocol is complete—since information dispersal itself requires only one initial round of messages from the dealer, the dealer's entire role is sending messages in the first round.

2) Share Verification: Each party $\mathcal{P}_i$ waits for ReliableBroadcast and Disperse to complete, and then retrieves their payload $\{z_i\}$. The party then attempts to decrypt and validate its shares. If decryption is successful and all the shares are valid, then $\mathcal{P}_i$ signals this by sending an `OK` message to the other recipients. The goal of the `OK` and `READY` messages (lines 302-307) is to ensure that if any party outputs a share, then enough correct parties have shares for share recovery to succeed if necessary.

3) Implicating a faulty dealer: If any honest party $\mathcal{P}_i$ receives a share that either fails to decrypt or fails verification, they reveal their secret key by sending $(\texttt{IMPLICATE}, SK_i, k)$ to all, which other parties can use to repeat the decryption and confirm that the dealer dispersed invalid data.

4) Share Recovery: If an honest party discovers their shares are faulty after other honest parties have already output, the protocol must enter Share Recovery. In this phase, parties with valid shares are presented with evidence that the dealer is faulty. If convinced, these parties will divulge the keys needed to decrypt their own shares. To avoid the need for constant re-keying, we present a practical modification for long-term keys in Section V-C.

B. Bandwidth-Optimized Failure Recovery

a) hbACSS1: Efficient Recovery for Additively-Homomorphic PolyCommits: When using an additively

Algorithm 1 $\mathsf{hbACSS0}(D, \mathcal{P}_1, ..., \mathcal{P}_N)$ for dealer D and parties $\mathcal{P}_1, ..., \mathcal{P}_N$

Setup:

1: Each party begins $\mathcal{P}_i$ with SK_i such that $\mathsf{PK}_i = g^{\mathsf{SK}_i}$
2: The set of all $\{\mathsf{PK}_j\}_{j\in[N]}$ are publicly known
3: Set up the polynomial commitment $\mathsf{SP} \leftarrow \mathsf{Setup}(1^\kappa, t)$

As dealer D with input $(s_1, ..., s_B)$:

// Secret Share Encoding
101: Sample B random degree-t polynomials $\phi_1(\cdot)...\phi_B(\cdot)$ such that each $\phi_k(0) = s_k$ and $\phi_k(i)$ is $\mathcal{P}_i$'s share of s_k

// Polynomial Commitment
102: $\mathbf{C} \leftarrow \{\mathsf{PolyCommit}(SP, \phi_k(\cdot))\}_{k\in[B]}$
103: $\mathsf{ReliableBroadcast}(\mathbf{C})$

// Encrypt and Disperse
104: $\{\pi_i\}_{i\in[1,N]} \leftarrow \mathsf{BatchProveEval}(SP, \mathbf{C}, \{\phi_k(\cdot)\}_{k\in[1,B]})$
105: **for** each $\mathcal{P}_i$ **do**
106: $\quad z_i \leftarrow \mathsf{Enc}_{\mathsf{PK}_i}(\pi_i \| \{\phi_k(i)\}_{k\in[1,B]})$
107: $\mathsf{Disperse}(\{z_i\}_{i\in[1,N]})$

As receiver $\mathcal{P}_i$:

// Wait for broadcasts
201: Wait to receive $\mathbf{C} := \{C_k\}_{k\in[B]} \leftarrow \mathsf{ReliableBroadcast}$
202: Wait for $\mathsf{Disperse}$ to complete

// Decrypt and validate
203: $z_i \leftarrow \mathsf{Retrieve}(i)$
204: $\{\phi_k(i)\}_{k\in[B]}, \pi_i \leftarrow \mathsf{Decrypt}_{\mathsf{SK}_i}(z_i)$
205: **if** $\mathsf{BatchVerifyEval}(\mathbf{C}, i, \{\phi_k(i)\}_{k\in[1,B]}, \pi_i) \neq 1$ or decryption fails **then**
206: $\quad$ **sendall** $(\texttt{IMPLICATE}, \mathsf{SK}_i)$
207: otherwise, valid shares are owned, so **sendall** `OK`

As receiver $\mathcal{P}_i$ (continued)

// Bracha-style agreement
301: On receiving `OK` from $2t + 1$ parties,
302: $\quad$ **sendall** `READY`
303: On receiving `READY` from $t + 1$ parties,
304: $\quad$ **sendall** `READY` (if haven't yet)
305: Wait to receive `READY` from $2t + 1$ parties,
306: $\quad$ **if** all owned shares are valid (line 207) **then**
307: $\quad\quad$ **output** shares $\{\phi_k(i)\}_{k\in[B]}$

// Handling Implication
401: On receiving $(\texttt{IMPLICATE}, \mathsf{SK}_j)$ from some $\mathcal{P}_j$,
402: $\quad$ ignore if already in *Share Recovery*
403: $\quad$ Discard if $\mathsf{PK}_j \neq g^{\mathsf{SK}_j}$
404: $\quad z_j \leftarrow \mathsf{Retrieve}(j)$
405: $\quad$ **if** $\mathsf{BatchVerifyEval}(\mathbf{C}, j, \{\phi_k(j)\}_{k\in[1,B]}, \pi_j) \neq 1$ or decryption fails **then**
406: $\quad\quad$ Proceed to *Share Recovery* below

// Share Recovery
501: **if** P_i previously output valid shares (line 307) **then** Multicast SK_i and return
502: Otherwise, on receiving SK_j from $\mathcal{P}_j$,
503: $\quad z_j \leftarrow \mathsf{Retrieve}(j)$
504: $\quad$ **if** $\mathsf{BatchVerifyEval}(\mathbf{C}, j, \{\phi_k(j), \pi_{j,k}\}_{k\in[B]})$ **then**
505: $\quad\quad$ Save $\{\phi_k(j)\}_{k\in[B]}$
506: On successfully verifying shares from $t + 1$ parties
507: $\quad$ Interpolate $\{\phi_k(i)\}_{k\in[B]}$ from valid shares
508: $\quad$ **output** shares $\{\phi_k(i)\}_{k\in[B]}$

homomorphic PolyCommit scheme (such as AMT proofs [57]), share recovery can be performed in a more bandwidth-efficient manner by utilizing the batch reconstruction technique of [11].

When instantiated with the KZG PolyCommit, our scheme achieves an improved asymptotic worst-case bandwidth over previous work, as shown in Table II. However, due to high real-world proof cost of the KZG scheme, we find that optimizing for computational efficiency is more important to achieve network scaling. Nonetheless, we present this optimization as potential future work.

We show the pseudocode for this optimization in Algorithm 2 and refer to the version of our ACSS scheme which utilizes additively homomorphic polynomial commitments in this way as hbACSS1. This algorithm specification assumes that a single evaluation can be checked without knowledge of other evaluations, but this is not strictly necessary (as we will later show).

In the first step of the new share recovery, parties wait for $t + 1$ `R1` messages from parties that received valid shares originally. These messages can be checked individually by making use of homomorphic properties of the PolyCommit scheme. Every correct party $\mathcal{P}_j$ participates in the second phase of share recovery by reconstructing one column of the bivariate polynomial $\phi(\cdot, j)$.

Algorithm 2 $\mathsf{hbACSS1ShareRecovery}(\mathcal{P}_1...\mathcal{P}_N)$ as party $\mathcal{P}_\mathsf{i}$

Let $\phi(x, y)$ be a degree t, t bivariate polynomial such that $\phi(i, k)$ gives $\mathcal{P}_\mathsf{i}$'s share of s_k
501: **for** each set of $t + 1$ secrets in B **do**
502: $\quad$ Interpolate $\{C_k\}_{k\in[N]}$ from $\{C_k\}_{k\in[t+1]}$
503: $\quad$ **if** we received valid shares (line 307) **then**
504: $\quad\quad$ Interpolate $\{\pi_{i,k}\}_{k\in[N]}$ from $\{\pi_{i,k}\}_{k\in[t+1]}$
505: $\quad\quad$ **for** each $\mathcal{P}_j$ **do**
506: $\quad\quad\quad$ **send** $(\texttt{R1}, \phi(i, j), \pi_{i,j})$ to $\mathcal{P}_j$
507: $\quad$ On receiving $(\texttt{R1}, \phi(k, i), \pi_{k,i})$ from $t+1$ parties such that $\mathsf{VerifyEval}(C_i, k, \phi(k, i), \pi_{k,i}) = 1$,
508: $\quad\quad$ Interpolate $\phi(\cdot, i)$
509: $\quad\quad$ **for** each $\mathcal{P}_j$ **do**
510: $\quad\quad\quad$ **send** $(\texttt{R2}, \phi(j, i))$ to $\mathcal{P}_j$
511: $\quad$ On receiving $(\texttt{R2}, \phi(i, k))$ from at least $2t+1$ parties,
512: $\quad\quad$ Robustly interpolate $\phi(i, \cdot)$
513: $\quad\quad$ **output** shares $\{\phi(i, k)\}_{k\in[t+1]}$

The second step is the transpose, where each party reconstructs the row polynomial corresponding to its shares. Since all correct parties send an `R2` message, even if they did not originally receive valid shares, we can interpolate through ordinary robust decoding rather than using the evaluation proofs.

b) hbACSS2: Efficient Recovery for any PolyCommits: We now describe hbACSS2, which achieves the same asymp-

totic behavior as hbACSS1 with non-homomorphic PolyCommits at the practical cost of a $\sim 3\times$ overhead to computation and bandwidth.

We again use the batch recovery protocol of [11]. Observe that this base protocol requires two robust polynomial interpolations (In R1 and R2). In Algorithm 2 we are able to make up for missing shares by utilizing homomorphic evaluation proofs to individually validate each point of R1, negating the need to rely on enough honest points to perform error correction.

Lacking this option, we can instead have the dealer provide the additional required information. We now require the dealer to deal $BN/(t+1)$ polynomials to share B secrets, leading to a $\sim 3\times$ overhead in the $N = 3t+1$ setting. In short, the dealer needs to interpolate polynomials $t+2$ through N for every batch of $t+1$ polynomials and provide the necessary proofs to replace lines 502 and 504 in Algorithm 2. We additionally can no longer assume that evaluation proofs are separable and instead require the dealer to split each recipient's proofs into N different batch-verifiable sets which themselves can be passed along into share recovery. We include a complete protocol description of hbACSS2 in the full version of this paper.

hbACSS2 may be desirable in systems that reach a level of scaling in which $O(N^2 \log(N))$ amortized network bandwidth is a significant hurdle or where DOS-like behaviour may be routine. We do note, however, that because all of our hbACSS protocols provide proof of Byzantine behavior (and share recovery is only necessary under Byzantine faults), combining hbACSS0 with malicious player eviction is likely to be more practical for many settings.

C. Long-Term Key Use

For simplicity, we described hbACSS using a generic public-key encryption scheme. While easier to explain, this would lead to suboptimal performance in practice due to (a) the high concrete complexity of public-key encryption and (b) the need to refresh keys after dealer implication. This can be ameliorated with a hybrid encryption scheme with long-term keys as follows:

At the start of the protocol, the dealer should create an ephemeral keypair $\{\mathrm{SK}_d, \mathrm{PK}_d = g^{\mathrm{SK}_d}\}$ and add g^{SK_d} to the ReliableBroadcast message of line 103 in Algorithm 1. Next, in line 106, the dealer should encrypt $\mathcal{P}_\mathsf{i}$'s message with a symmetric encryption scheme, using the shared key $\mathrm{K}_d^i = \mathrm{PK}_i^{\mathrm{SK}_d}$. Lastly, in the implicate phase, instead of sending SK_i, $\mathcal{P}_\mathsf{i}$ should instead send K_d^i and the zero knowledge proof $\mathrm{NIZK}\{(\mathrm{SK}_i) : g^{\mathrm{SK}_i} = \mathrm{PK}_i \wedge \mathrm{PK}_d^{\mathrm{SK}_i} = \mathrm{K}_d^i\}$.

In this way, we avoid the need to use a different PKI for every hbACSS instance, so long as said PKI is not used in a different protocol which could compromise the secret keys. These modifications apply for every version of our scheme.

D. Security Analysis of hbACSS

Theorem 1. *The hbACSS protocol (Algorithm 1) satisfies the requirements of an ACSS protocol (with high probability) when instantiated with a polynomial commitment scheme* $(\mathsf{Setup}, \mathsf{PolyCommit})$*, an AVID protocol* $(\mathsf{Disperse}, \mathsf{Retrieve})$*, a reliable broadcast protocol* ReliableBroadcast*, and a semantically public key encryption scheme* $(\mathsf{Enc}, \mathsf{Dec})$ *with a pre-established PKI such that each party* $\mathcal{P}_i$ *knows their secret key* SK_i *and the public keys* $\{\mathrm{PK}_i = g^{\mathrm{SK}_i}\}_{i\in[N]}$ *are known.*

Proof: **Correctness.** The correctness property follows easily: If the dealer D is correct, then ReliableBroadcast and Disperse complete, so each honest party receives their valid shares and outputs them through the ordinary case (line 307).

Secrecy. We will first consider an honest dealer's share secrecy when the Share Recovery phase is not invoked (for simplicity, we will refer to hbACSS with a batch size of 1). We do not assume any secrecy property in our AVID protocol, and therefore assume our adversary can see every message sent by the dealer: $\{\mathsf{Enc}_{\mathsf{PK}_i}(\pi_i \| \{\phi_k(i)\}_{k\in[1,B]})\}_{i\in[1,N]}$, along with the broadcasted polynomial commitment, the PKI, and its t decryption keys.

We now design a simulator which can simulate the adversary's view. The simulator receives as input the t polynomial shares belonging to corrupted parties. The simulator chooses a polynomial $\hat{\phi}$ that is consistent with these t shares, and for simplicity fixes the remaining degree of freedom such that $\hat{\phi}(0) = 0$. The simulator creates an honest commitment to $\hat{\phi}$ and creates t honest evaluation proofs, and encrypts these. Lastly, the simulator generates the adversary's keypairs, fills the rest of the PKI with random strings, and generates the remaining ciphertexts by encrypting zero-strings. More formally, our indistinguishability argument is:

Real World:

$$\left\{\begin{array}{l} \mathsf{SP} \leftarrow \mathsf{Setup}(1^\kappa, t) \\ C \leftarrow \mathsf{PolyCommit}(\mathsf{SP}, \phi(\cdot)) \\ \mathsf{PK}_{i\in[1..N]}, \mathsf{SK}_{i\in[1..t]} \leftarrow \mathsf{GenPKI}(1^\kappa, N) \\ \{\mathsf{Enc}_{\mathsf{PK}_i}((i, \phi(i), \pi_i) \leftarrow \mathsf{ProveEval}(\mathsf{SP}, \phi(i), \mathsf{aux})\}_{i\in N} \end{array}\right\}$$

Ideal World:

$$\left\{\begin{array}{l} \mathsf{SP} \xleftarrow{\$} \mathsf{Setup}(1^\kappa, t) \\ C \leftarrow \mathsf{PolyCommit}(\mathsf{SP}, \phi(i)_{i\in[1..t]} \bigcup \hat{\phi}(0) \xleftarrow{\$} Z_p^*) \\ \mathsf{PK}_{i\in[1..t]}, \mathsf{SK}_{i\in[1..t]} \xleftarrow{\$} \mathsf{GenPKI}(1^\kappa, t) \\ \mathsf{PK}_{i\in[t+1..N]} \xleftarrow{\$} \mathcal{U} \\ \{\mathsf{Enc}_{\mathsf{PK}_i}((i, \hat{\phi}(i), \pi_i) \leftarrow \mathsf{ProveEval}(\mathsf{SP}, \hat{\phi}(i), \mathsf{aux})\}_{i\in[1..t]} \\ \{\mathsf{Enc}_{\mathsf{PK}_i}(0^*)\}_{i\in[t+1..N]} \end{array}\right\}$$

In particular, a simulator algorithm that only knows t, N, and the adversary's t shares can create a protocol view that is computationally indistinguishable from the adversary's view in hbACSS, thus proving that our protocol leaks no additional information about ϕ.

The other option is for an adversary to attempt to initiate Share Recovery. However, to get an honest party to divulge their share, an adversary must present them with a valid Implication proof. The *correctness* property of the AVID scheme ensures that honest parties will retrieve the correct encrypted shares. Consequently, the use of an incorrect SK will be rejected (line 402), whereas a correct SK will retrieve the untampered message, which will verify given an honest dealer.

Agreement. It is easy to check that parties only output shares that are consistent with the broadcasted polynomial

Protocol	Best Case	Crashes	Byzantine
hbACSS0 +HBPC	$O(N \log N)$	$O(N \log N)$	$O(N^2 \log N)$
hbACSS2 +HBPC	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
hbACSS1 +KZG[42]	$O(N)$	$O(N)$	$O(N)$
hbACSS1 +AMT[57]	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
VSS-R[9]	$O(N)$	$O(N^2)$	$O(N^2)$
eAVSS-SC[7]	$O(N^2)$	$O(N^2)$	$O(N^2)$
AVSS[22]	$O(N^3)$	$O(N^3)$	$O(N^3)$

TABLE II: Amortized whole-network bandwidth of AVSS protocols with optimal resilience and share recovery

commitments. The challenge is in showing that if any correct party outputs a share, then all of them do. In the following, assume a correct party has output a share, either through the typical path (line 307) or through share recovery (line 508). In either case, broadcast and dispersal must have completed and the party must have received $2t+1$ `READY` messages (line 305).

First, notice the `READY`-amplification in line 304 plays the same role as in Bracha broadcast:

Claim 1. *If any correct party outputs a share, then all correct parties eventually receive* $2t + 1$ `READY` *messages (line 305).*

If any correct party outputs on line 307, they received $2t + 1$ `READY` messages, meaning at least $t + 1$ correct parties must have sent `READY` messages, which causes all correct parties to send `READY` messages.

If any correct party outputs on line 508, they must have received a share from another correct party which received $2t + 1$ `READY` messages, per the check on line 501.

Next, the following claim ensures that share recovery can proceed if necessary:

Claim 2. *If any correct party outputs a share, then at least* $t + 1$ *correct parties receive valid shares.*

For `READY`-amplification to begin, some correct party must have initially sent `READY` after receiving $2t + 1$ `OK` messages (line 302), thus $t + 1$ correct parties must have successfully received valid shares (line 207).

Because of the availability and agreement properties of dispersal, every correct party either receives valid shares (and by then Claim 1 outputs ordinarily) or else receives an invalid share and initiates share recovery, which by Claim 2 is able to proceed. ■

E. Performance Analysis of hbACSS

Although instantiating hbACSS with KZG PolyCommits sets a new asymptotic bandwidth record, we focus our analysis on hbPolyCommit, which we believe to be more practical due to lower computation costs and the lack of a trusted setup. Still, we summarize some notable asymptotic results relating various hbACSS options with other works in Table II.

1) AVID Costs: In order to allow Byzantine parties to always be identified (a property we refer to as *non-repudiation*), we rely on Asynchronous Verifiable Information Dispersal (AVID). To send a message of size M, our implementation of AVID in the $t < \lfloor N/3 \rfloor$ setting incurs a bandwidth overhead of $\sim 3M + N \log(N)$ for the sender and receiver, and an overhead of $\sim M/t + \log(N)$ to be sent and received by each non-receiving participant. The logarithmic factors of this overhead are due to the sending of Merkle branches to validate erasure-codings. We remark that this cost does not depend on the size of M and is consequently amortized away with a sufficiently large message. Computational costs related to Merkle trees are similarly amortized.

In hbACSS, the dealer must use AVID to send N messages of size M, which incurs a constant factor overhead of $3NM$ provided that the messages are at least $O(N \log(N))$-sized, which is the case given a batch size B of at least $O(N)$ and logarithmically-sized evaluation proofs. For the purposes of this analysis, we will be operating under those two assumptions and therefore write the total dealer bandwidth as $\sim 3NB \log N$.

A recipient must use AVID to retrieve one M-sized message and aid in the retrieval of N-1 other M-sized messages. In hbACSS0 and hbACSS1, provided that M is of size $B \log(N)$ as before, this leads to an amortized overhead of $\sim 3B \log(N)$ sent and $\sim 6B \log(N)$ received. In the case of hbACSS2, these become $\sim 9B \log(N)$ and $\sim 18B \log(N)$ respectively.

2) PolyCommit Costs: The computational costs associated with our ACSS scheme are dominated by the costs of proving and verifying polynomial evaluations through a PolyCommit scheme. While many suitable schemes could be used to instantiate hbACSS, here we will analyse the computation involved when the hbPolyCommit scheme is chosen.

The prover's most asymptotically significant computations are the matrix multiplications required to compute c_L and c_R. These result in $2t + 2$ field integer multiplications and $2t$ additions for every proof in each batch. Thus, the total number of required field integer operations is $\sim 4BNt$ ($\sim 12BNt$ for hbACSS2) to calculate BN proofs, leading to an amortized dealer computation complexity of $O(N)$ per proof. We note that the $O(\log(N))$ group operations per proof are more significant at the levels of N where we evaluate our protocol, as we show by achieving similar performance to a scheme with $O(\log(N))$ amortized computation in Section VII.

For the verifier, there is no similar $O(N)$ operation that must be performed for each proof in the batch. Instead, the most significant part of the verifier's work is exponentiating the group elements L and R at each recursive step of the protocol for each proof. This leads to an amortized $O(\log(N))$ verifier computation per proof in B.

3) Overall Best Case Performance: hbACSS has favorable performance characteristics when equipped with a PolyCommit scheme that utilizes batch amortization. In the best case, the dealer incurs $\sim 3NB \log(N)$ bandwidth overhead in sending B logarithmically-sized proofs to N different receivers. The dealer performs $O(N)$ computation per proof in field integer operations, and at most $O(\log(N))$ computation for all other operations. This does not change, regardless of the failure model.

In the best case, recipients receive and verify B evaluation proofs and aid in the retrieval of an additional NB proofs. The amortized computation cost is then $O(\log(N))$ while the bandwidth is $\sim 3B \log(N)$ sent and $\sim 6B \log(N)$ received. All of these costs increase by a constant factor of 3 in hbACSS2.

4) Crash Fault Performance: As a fully asynchronous protocol, the performance of hbACSS is minimally affected

by crashed nodes, network delays, and partitions. The protocol will progress as quickly as computation and the network allow.

5) Byzantine Fault Performance: If the dealer is Byzantine, we use our Share Recovery protocol satisfy our *agreement* property: if some honest parties output valid shares, all honest parties do. In hbACSS0, those without valid shares initially need to download at least t+1 batches of shares and run BatchVerifyEval on each of them. When using hbPolyCommit, this requires an additional $O(BN\log(N))$ bandwidth and $O(BN\log(N))$ computation to obtain B valid shares, while honest parties with valid shares incur an additional $O(B\log(N))$ computation to check an implication and $O(BN\log(N))$ bandwidth in assisting AVID retrievals.

In contrast, hbACSS1 and hbACSS2 use a more efficient share recovery mechanism. In the first round, each honest party with valid shares sends $O(B\log(N))$ information, followed by all parties sending $O(B)$ in the second round. The overall amortized network bandwidth is thus still $O(N\log(N))$. Computationally, nodes need to validate an additional $B(t+1)/N$ shares in round one, but otherwise only need to perform polynomial interpolations and evaluations.

A Byzantine recipient can attempt to interfere with the AVID protocol, inaccurately send OK and READY messages, and attempt to falsely implicate the dealer. Of these, none impact the security of the protocol and only false implications affect protocol performance. A Byzantine adversary controlling t participants could initiate t false IMPLICATE messages, causing honest parties to retrieve $O(BN\log(N))$ additional data and run BatchVerifyEval t times.

The Byzantine recipient case can be mitigated against by using even larger batches: instead of using one $O(N)$-sized batch, the dealer should deal N different $O(N)$-sized sub-batches, for a total batch size of $O(N^2)$. Each sub-batch is encrypted and dispersed individually and the IMPLICATE message is modified to include a block number where an invalid proof can be found. With this modification, the amortized computation and communication complexity of an honest party who receives valid shares is the same as in the best case.

We remark that although our performance can be worsened by a Byzantine adversary, every Byzantine action leaves behind a proof of the adversary's malfeasance. Since Byzantine adversaries can be identified and excluded from subsequent protocol invocations, we do not feel that a performance degradation due to Byzantine behavior is a significant drawback of our protocol.

6) Recommendations on hbACSS Version Usage: Each version of hbACSS has its own strengths and weaknesses. hbACSS0 and hbACSS2 accommodate the widest variety of polynomial commitment schemes, hbACSS0 and hbACSS1 have the best general performance in situations with minimal byzantine faults, and hbACSS1 and hbACSS2 best handle share recovery in the case of many faults.

Generally, we feel that hbACSS0 is the best choice in the widest variety of applications: all hbACSS versions identify byzantine players and we suspect that most applications would not allow byzantine players to participate in the protocol indefinitely, rendering the relatively inefficient share recovery to be of minimal concern. As N grows, however, hbACSS1 may become more preferable for better bandwidth and computation asymptotics when byzantine faults do occur.

However, a more efficient byzantine failure recovery may be desirable in decentralized applications where player churn is high and the barrier to entry is low, or alternatively in applications where the number of players is large enough that any $\Omega(N^2)$ bandwidth operation is infeasible. In these situations hbACSS1 is likely preferable if it can accommodate a fast polynomial commitment scheme with tolerable setup and hardness assumptions, as it has ∼3x smaller constants than hbACSS2. In contrast, hbACSS2 has a higher base overhead but is less affected by DOS attempts and may be preferable for that reason.

VI. Using hbAVSS for Robust Input to MPC

Recent work [47] has explored the practicality of robust and asynchronous MPC, but falls short of full robustness, instead dividing itself into a robust *online* phase which responds to client inputs and a non-robust *offline* phase which performs input-independent precomputation. One such precomputation is generating randomized input masks: unknown random values shared amongst the servers to facilitate easy client input. When a client wishes to contribute an input m, servers send the client their shares of the mask r, allowing the client to privately reconstruct it. The client then broadcasts $(r+m)$ and servers locally compute $[\![m]\!] = (r+m) - [\![r]\!]$.

Beerliová-Trubíniová, and Hirt [11] introduced a method to compute secret-shared random values by using hyper-invertible matrices to create linear combinations of (non-verifiably) secret shared locally-random values. However, as this protocol lacks non-repudiation in the case of faults, its player elimination protocol proceeds two at a time (where only one eliminated player is guaranteed to have been malicious), making it unsuitable for an asynchronous, optimally fault tolerant setting.

The use of a complete AVSS scheme bypasses this issue entirely, as it guarantees that all honest parties hold well-formed shares and no further integrity checking is needed. As the previous integrity check involved opening some random values, we also improve input mask yield. Provided that $N - t \leq j \leq N$ parties secret share a random input, the yield y is given by $j - t$. An asynchronous common subset protocol can be used to agree on a j-sized set of inputs to use, provided parties have access to a random beacon.

Regardless of whether a client uses input masks or acts as an AVSS dealer to instantiate its inputs, we stress that using an AVSS with recoverable shares is essential. Once the MPC is underway, the shares $[\![m]\!]$ may be used in arithmetic circuits, depending on the computation. Further, it may be necessary to reconstruct linear combinations of $[\![m]\!]$ along with inputs contributed by other parties.

In our setting, we expect MPC servers to be fairly long-lasting entities. Because hbACSS provides non-repudiation, any server acting maliciously will be caught and should be excluded from subsequent rounds. Consequently, we expect Byzantine faults to be minimal and therefore use hbACSS0 for our application as it offers the best performance in scenarios where such faults are uncommon. We evaluate the performance of our solution in Section VII.

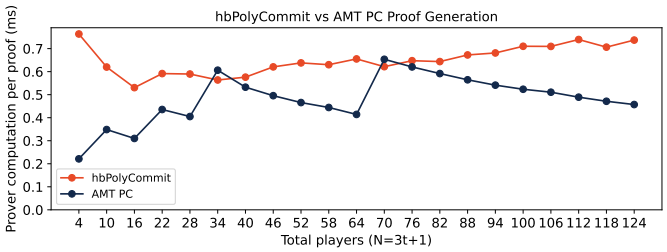

Fig. 2: Amortized proof generation times

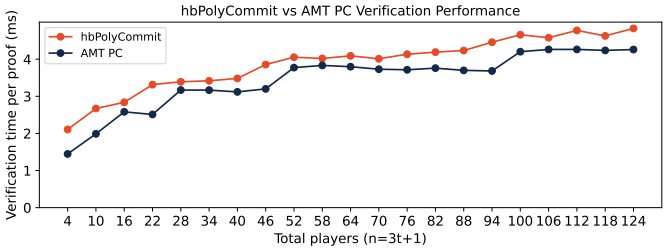

Fig. 3: Amortized proof verification times

VII. Implementation and Evaluation

In this section, we evaluate hbPolyCommit and hbACSS in the $N = 3t + 1$ setting. In particular, we directly compare hbPolyCommit with the state-of-the-art AMT polynomial commitments, demonstrate baseline costs of each of our protocols, show full end-to-end costs of hbACSS0 and hbACSS2 when instantiated with hbPolyCommit, and demonstrate the resulting MPC input mask yield.

a) Experimental Setup: We implemented[1] hbPolyCommit and hbACSS primarily in Python, using a wrapper for the Zcash team's Rust pairing library [41], which implements the bls12-381 elliptic curve. Our benchmarks are all single-threaded and done inside a docker container for easy replicability. Since the original AMT VSS benchmarks run on Linux, we created a docker image for the AMT benchmarks so that container-induced overhead is consistent. We also modified their code so that it would run in the $N = 3t + 1$ setting. Both containers run on our experiment machine which has an Intel Xeon E5-2620 v4 CPU and 128GB RAM.

All benchmarks are given as a per-share per-user measurement. Unless otherwise specified, our benchmarks are run on a batch of $\sim 2N$ shares, as we found this sufficient to demonstrate asymptotic behavior. All experiments were run using pytest-benchmark and were run at least three times.

b) Limitations: We believe the code can be optimized by moving more of it to Rust and optimizing some algorithms, such as polynomial interpolation. Our evaluation does not account for network latency as we estimate network effects to be relatively insignificant. However, we still serialize messages and use Asyncio queues to approximate I/O between tasks.

A. Polynomial Commitment Evaluations

a) Proof generation time: In Figure 2, we show the per-proof generation time of AMT PolyCommits and hbPolyCommit. Asymptotically, AMT's $O(\log(N))$ amortized per-proof generation time is better than hbPolyCommit's $O(N)$ time, but the practical gap between them is small since the only linear component of hbPolyCommit is cheap field operations. From the graph, the reference implementation for AMTs shows a spike in costs whenever N crosses a power of 2, whereas hbPolyCommit shows a jagged-but-gradual increase in cost as the amount of work required varies based on the number of 1's in the bit decomposition of $t + 1$.

[1] code available at github.com/tyurek/hbACSS

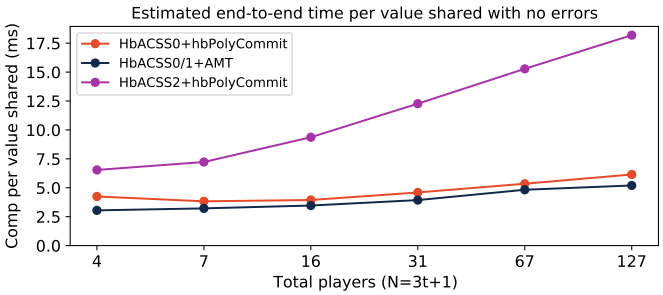

Fig. 4: Estimated end-to-end time per party per value shared when all shares are correct

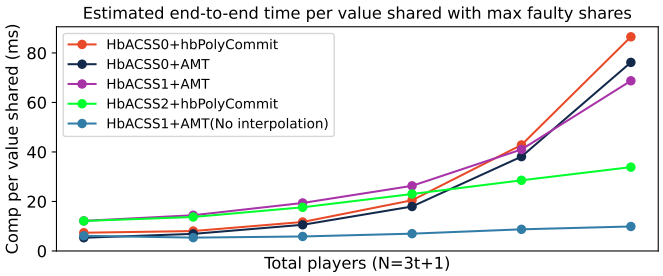

Fig. 5: Estimated end-to-end time per party per value shared when t invalid shares are dealt

b) Verification time: In Figure 3, we show the per-proof verification times of both PolyCommit schemes. Both schemes show a similar $O(\log N)$ performance in practice. hbPolyCommit is jagged for the same reasons as before, while AMT exhibits spikes when t crosses a power of 2.

c) Proof size: Given N=3t+1 and assuming one field element takes 32 bytes, hbPolyCommit roughly needs 32+($\lceil\log(t)\rceil$+1)*2*32 bytes amortized per proof. In practice, the length varies slightly since our hbPolyCommit construction handles odd and even N differently at each recursive step. AMT PolyCommits need 32+($\lfloor\log(t+1)\rfloor$+1)*32 bytes per proof. Though hbPolyCommit proofs are slightly larger, both exhibit the same asymptotic behavior.

B. hbACSS Evaluations

We implement our ACSS protocols and evaluate them in two scenarios: zero faults and t dealer-injected faults (the most in which the protocol will still complete). We avoid evaluating Byzantine receivers as all they can do to slow down the protocol is send false `IMPLICATE` messages, which are easily detected and can be mitigated by more batching (as in Section V-E5). Moreover, DOS-like behavior could easily be done out-of-band.

Although we do reimplement AMTs in our framework, we use the performance of their reference implementation to simulate end-to-end hbACSS evaluation times, thus avoiding bias from our reimplemention of their protocol being underoptimized. However, as the cost of interpolating commitments and evaluation proofs is not benchmarked in the AMT reference

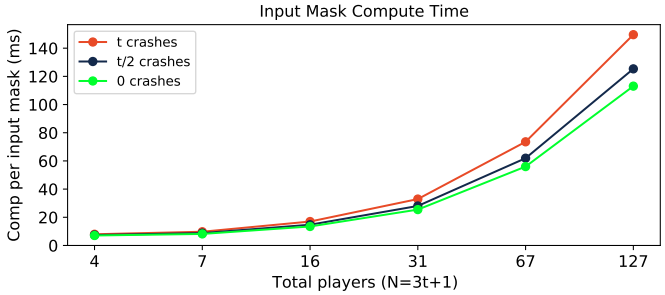

Fig. 6: Computational work per node to generate an input mask with a varying number of crash faults

implementation, we measure this in our framework to estimate the costs of share recovery in hbACSS1.

Figure 4 shows our fault-free performance for hbACSS0 instantiated with both hbPolyCommit and AMTs as well as hbACSS2 with hbPolyCommit. hbACSS1 is merged with hbACSS0 in this figure, as the protocols are equivalent in the fault-free case. To see the amortization benefits of hbACSS2, we ran it with a batch size of $\sim N^2$ (as necessitated by design).

These trends change in Figure 5 where we evaluate the end-to-end performance when the dealer sends faulty shares to t different verifiers. Since hbACSS0 needs to verify asymptotically more evaluation proofs under maximal Byzantine faults, its performance relative to hbACSS2 decays as N grows larger. While hbACSS1 grows quickly in this figure, the cost is almost entirely attributable to the computational overhead of interpolating evaluation proofs (we add a line in which commitment and evaluation proof interpolation costs are removed to illustrate this point). These costs would be greatly reduced if AMT evaluation proofs were constant-sized.

While our figures do not include latency and bandwidth, we estimate that they do not significantly impact our protocol's performance: At the time of this writing, Amazon Web Services (AWS) has a typical 206 ms inter-region latency between North Virginia and Sydney [56], [6] with roughly 55 Mbps of bandwidth [35]. hbACSS has a constant round complexity of four one-way trips in the fault-free case (two for ReliableBroadcast and AVID in parallel, one for the OK messages and one for the READY messages), resulting in roughly 0.8 seconds of cumulative latency per batch.

When N=127 (the largest case we evaluate), the hbPolyCommit evaluation proofs are roughly 416 bytes each, which is far larger than anything else sent in hbACSS. Nonetheless, after applying the roughly $6x$ bandwidth penalty for our method of distributing payloads, recipients could still receive thousands of shares per second with our scheme if computation was not an issue.

C. Input Mask Generation

In Section VI we showed how ACSS can be used to robustly generate input masks in an asynchronous setting. Moreover, the number of input masks yielded is linear in the number of total shares dealt. We demonstrate this in Figure 6: even with t crash faults, the total per-node computation required is only moderately affected, with roughly 30% overhead for larger values of N. In the case where all simulated servers ran a hbACSS instance, we saw yields of ~39 input masks per second when $N = 31$ and ~9 per second at $N = 127$.

D. Further Applications

We note that while this paper primarily focuses on one particular MPC use-case, the tools that we have developed here are more broadly useful in the MPC space. For example, many asynchronous MPC protocols [10], [54], [52], [32], [33] rely on AVSS for robustness, which usually ends up as the primary bottleneck. The recent work of [31] set a new network bandwidth record in optimally-resilient information theoretic ACSS of $O(N^3)$, which when combined with the techniques of [33] achieved an asynchronous MPC with $O(N^4)$ bits of bandwidth per multiplication gate. By using the same techniques from [33] and switching to a computational adversary, our work could result $O(N^2)$ bits per multiplication gate given trusted setup and $O(N^2 \log N)$ bits without trusted setup.

Another interesting application of our protocols can be MPC-based anonymous broadcast [47], [2], [34] that allows a set of clients to send their messages together such that no individual message from the broadcast output set can be linked to its sender client. For example, PowerMix [47] employs input masks as described above for sending client messages m_i to their MPC system that first securely computes powers $m_i^2, m_i^3, \ldots$ and then computes Newton's power sums. Here, our batched ACSS offers an interesting other alternative: every client itself can send its messages and powers $m_i, m_i^2, m_i^3, \ldots$ using batched ACSS such that the MPC is reduced to performing secure addition towards creating Newton's power sums. Batched ACSS can offer a linear factor improvement over previous ACSS/AVSS protocols here.

We also remark that our protocols could prove useful in the construction of a randomness beacon (an input mask is merely a piece of public randomness), a problem which has seen a surge of recent interest [15], [37], [27]. However, we leave such applications to future work.

VIII. Conclusion

In this paper, we took a significant step towards closing the gap between theory and practice in robust and asynchronous multiparty computation. We designed and implemented a batch-efficient ACSS scheme which itself utilizes batch-efficient polynomial commitment schemes. We created the first optimally-resilient ACSS protocol with amortized linear network overhead along the way, but focused our attention on computational hurdles and trusted setup assumptions that stand as a barrier to practical deployment (while still achieving new asymptotic goals). To that end, we developed a polynomial commitment protocol which performs comparably to peer constructions while removing the need for trusted setup. Lastly, we demonstrated the utility of our constructions in robustly generating MPC input masks.

IX. Acknowledgements

We thank our shepherd, Mauro Conti, and the anonymous reviewers for their helpful feedback. This work has been partially supported by the National Institute of Food and Agriculture (NIFA) under grant 2021-67021-34251, the National Science Foundation (NSF) under grants CNS-1846316 and 1943499, C3SR, and by IC3 industry partners.

REFERENCES

[1] I. Abraham, D. Dolev, and G. Stern. Revisiting asynchronous fault tolerant computation with optimal resilience, 2020.

[2] I. Abraham, B. Pinkas, and A. Yanai. Blinder – scalable, robust anonymous committed broadcast. In *ACM Conference on Computer and Communications Security*, 2020.

[3] N. Alexopoulos, A. Kiayias, R. Talviste, and T. Zacharias. Mcmix: Anonymous messaging via secure multiparty computation. In *USENIX Security Symposium*, 2017.

[4] N. AlHaddad, M. Varia, and H. Zhang. High-threshold avss with optimal communication complexity.

[5] A. Aly, E. Orsini, D. Rotaru, N. P. Smart, and T. Wood. Zaphod: Efficiently combining lsss and garbled circuits in scale. In *ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2019.

[6] A. AWS. Regions and zones. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html, 2021. Online; accessed 2021.

[7] M. Backes, A. Datta, and A. Kate. Asynchronous computational vss with reduced communication complexity. In *Cryptographers' Track at the RSA Conference*, 2013.

[8] J. Baron, K. El Defrawy, J. Lampkins, and R. Ostrovsky. How to withstand mobile virus attacks, revisited. In *ACM symposium on Principles of distributed computing*, 2014.

[9] S. Basu, A. Tomescu, I. Abraham, D. Malkhi, M. K. Reiter, and E. G. Sirer. Efficient verifiable secret sharing with share recovery in bft protocols. In *ACM Conference on Computer and Communications Security*, 2019.

[10] Z. Beerliová-Trubíniová and M. Hirt. Simple and efficient perfectly-secure asynchronous mpc. In *International Conference on the Theory and Application of Cryptology and Information Security*, 2007.

[11] Z. Beerliová-Trubíniová and M. Hirt. Perfectly-secure mpc with linear communication complexity. In *Theory of Cryptography Conference*, 2008.

[12] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM conference on Computer and communications security*, 1993.

[13] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *ACM Symposium on Theory of Computing*, 1988.

[14] M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous secure computations with optimal resilience. In *ACM symposium on Principles of distributed computing*, 1994.

[15] A. Bhat, N. Shreshta, A. Kate, and K. Nayak. Randpiper – reconfiguration-friendly random beacons with quadratic communication. Cryptology ePrint Archive, Report 2020/1590, 2020.

[16] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2016.

[17] J. Bootle and J. Groth. Efficient batch zero-knowledge arguments for low degree polynomials. In *IACR International Workshop on Public Key Cryptography*, 2018.

[18] S. Bowe, A. Gabizon, and M. D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. In *International Conference on Financial Cryptography and Data Security*, 2018.

[19] S. Bowe, A. Gabizon, and I. Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017.

[20] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 1985.

[21] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE Symposium on Security and Privacy (SP)*, 2018.

[22] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *ACM Conference on Computer and Communications Security*, 2002.

[23] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In *IEEE Symposium on Reliable Distributed Systems*, 2005.

[24] J. Camenisch and M. Stadler. Efficient group signature schemes for large groups. In *Annual International Cryptology Conference*, 1997.

[25] R. Canetti and T. Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *ACM Symposium on Theory of Computing*, 1993.

[26] J. Cartlidge, N. P. Smart, and Y. Talibi Alaoui. Mpc joins the dark side. In *ACM Asia Conference on Computer and Communications Security*, 2019.

[27] I. Cascudo and B. David. Albatross: publicly attestable batched randomness based on secret sharing. Cryptology ePrint Archive, Report 2020/644, 2020.

[28] M. Chen, C. Hazay, Y. Ishai, Y. Kashnikov, D. Micciancio, T. Riviere, A. Shelat, M. Venkitasubramaniam, and R. Wang. Diogenes: Lightweight scalable rsa modulus generation with a dishonest majority. *IACR Cryptol. ePrint Arch.*, 2020.

[29] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2020.

[30] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *IEEE FOCS*, 1985.

[31] A. Choudhury. Optimally-resilient unconditionally-secure asynchronous multi-party computation revisited. Cryptology ePrint Archive, Report 2020/906, 2020.

[32] A. Choudhury, M. Hirt, and A. Patra. Asynchronous multiparty computation with linear communication complexity. In *International Symposium on Distributed Computing*, 2013.

[33] A. Choudhury and A. Patra. An efficient framework for unconditionally secure multiparty computation. *IEEE Transactions on Information Theory*, 2016.

[34] H. Corrigan-Gibbs, D. I. Wolinsky, and B. Ford. Proactively accountable anonymous messaging in verdict. In *USENIX Security Symposium*, 2013.

[35] B. Cutler. Examining cross-region communication speeds in aws. https://medium.com/slalom-technology/examining-cross-region-communication-speeds-in-aws-9a0bee31984f, 2021. Online; accessed 2021.

[36] I. Damgård, Y. Ishai, M. Krøigaard, J. B. Nielsen, and A. Smith. Scalable multiparty computation with nearly optimal work and resilience. In *Annual International Cryptology Conference*, 2008.

[37] S. Das, V. Krishnan, I. M. Isaac, and L. Ren. Spurt: Scalable distributed randomness beacon with transparent setup. Cryptology ePrint Archive, Report 2021/100, 2021.

[38] D. Demmler, T. Schneider, and M. Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[39] S. Duan, M. K. Reiter, and H. Zhang. Beat: Asynchronous bft made practical. In *ACM Conference on Computer and Communications Security*, 2018.

[40] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Annual Symposium on Foundations of Computer Science*, 1987.

[41] J. Grigg and S. Bowe. pairing. https://github.com/zkcrypto/pairing.

[42] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, 2010.

[43] M. Keller. Mp-spdz: A versatile framework for multi-party computation. *IACR Cryptol. ePrint Arch.*, 2020.

[44] E. Kokoris Kogias, D. Malkhi, and A. Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[45] Y. Lindell. Parallel coin-tossing and constant-round secure two-party computation. *Journal of Cryptology*, 2003.

[46] Y. Lindell and S. Ranellucci. Unbound blockchain-crypto-mpc library white paper. https://github.com/unbound-tech/blockchain-crypto-mpc/

blob/master/docs/Unbound\%20Cryptocurrency\%20Wallet\%20Library\%20White\%20Paper.pdf, 2019.

[47] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. Miller. Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication. In *ACM Conference on Computer and Communications Security*, 2019.

[48] S. K. D. Maram, F. Zhang, L. Wang, A. Low, Y. Zhang, A. Juels, and D. Song. Churp: Dynamic-committee proactive secret sharing. In *ACM Conference on Computer and Communications Security*, 2019.

[49] F. Massacci, C. N. Ngo, J. Nie, D. Venturi, and J. Williams. Futuresmex: secure, distributed futures market exchange. In *IEEE Symposium on Security and Privacy (SP)*, 2018.

[50] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of bft protocols. In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[51] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. Cryptology ePrint Archive, Report 2011/091, 2011.

[52] A. Patra, A. Choudhary, and C. P. Rangan. Efficient statistical asynchronous verifiable secret sharing with optimal resilience. In *International Conference on Information Theoretic Security*, 2009.

[53] A. Patra, A. Choudhury, and C. P. Rangan. Communication efficient perfectly secure vss and mpc in asynchronous networks with optimal resilience. In *International Conference on Cryptology in Africa*, 2010.

[54] A. Patra, A. Choudhury, and C. P. Rangan. Efficient asynchronous verifiable secret sharing and multiparty computation. *Journal of Cryptology*, 2015.

[55] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual international cryptology conference*, 1991.

[56] A. Systems. Aws inter region latency. https://docs.aviatrix.com/HowTos/inter_region_latency.html, 2021. Online; accessed 2021.

[57] A. Tomescu, R. Chen, Y. Zheng, I. Abraham, B. Pinkas, G. G. Gueta, and S. Devadas. Towards scalable threshold cryptosystems. In *IEEE Symposium on Security and Privacy (SP)*, 2020.

[58] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. Doubly-efficient zksnarks without trusted setup. In *IEEE Symposium on Security and Privacy (SP)*, 2018.

[59] Y. Zhang, A. Steele, and M. Blanton. Picco: a general-purpose compiler for private distributed computation. In *ACM conference on Computer & communications security*, 2013.

APPENDIX

A. Definitions

Definition 6. *Discrete Log Relation Assumption. For all probabilistic, polynomial-time adversaries $\mathcal{A}$ and for all $n \geq 2$ there exists a negligible function* $\text{negl}(\kappa)$ *such that*

$$\Pr\left[\begin{array}{l}\mathbb{G} = \text{Setup}(1^\kappa), g_1, \ldots, g_n \xleftarrow{\$} \mathbb{G} \\ a_1, \ldots, a_n \in \mathbb{Z}_p \leftarrow \mathcal{A}(\mathbb{G}, g_1, \ldots, g_n) : \\ \exists a_i \neq 0 \wedge \prod_{i=1}^n {g_i}^{a_i} = 1\end{array}\right] \leq \text{negl}(\kappa)$$

We say $\prod_{i=1}^n {g_i}^{a_i} = 1$ is a non-trivial discrete log relation between the generators $g_1, \ldots, g_n$.

Definition 7. *Zero-Knowledge Argument of Knowledge. An argument of knowledge for $R[\mathsf{crs}]$ consists of three probabilistic polynomial-time algorithms: the prover $\mathcal{P}$, verifier $\mathcal{V}$, and* $\mathsf{Gen}(1^\kappa)$ *which outputs a common reference string* crs*. The transcript produced by $\mathcal{P}$ and $\mathcal{V}$ when interacting on inputs s and t is denoted by $tr \leftarrow \langle \mathcal{P}(\mathsf{crs}, s), \mathcal{V}(\mathsf{csr}, t)\rangle$. If $\mathcal{V}$ outputs 1 we write $\langle \mathcal{P}(\mathsf{crs}, s), \mathcal{V}(\mathsf{crs}, t)\rangle = 1$ and say that tr is accepting. Similarly, if $\mathcal{V}$ outputs 0 we write $\langle \mathcal{P}(\mathsf{crs}, s), \mathcal{V}(\mathsf{crs}, t)\rangle = 0$ and say that tr is rejecting.*

Definition 8. *Perfect Special Honest-Verifier Zero-Knowledge (PSHVZK). There exists a polynomial-time simulator $\mathcal{S}$ such that for every $\kappa \in \mathbb{N}$, every* crs *in the support of* $\mathsf{Gen}(1^\kappa)$*, every* $(\mathsf{stmt}, w) \in R[\mathsf{crs}]$ *and every z,*

$$\{\langle \mathcal{P}(\mathsf{crs}, \mathsf{stmt}, w), \mathcal{V}(\mathsf{crs}, \mathsf{stmt}; z)\rangle\} \equiv \{\mathcal{S}(\mathsf{crs}, \mathsf{stmt}, z)\}.$$

B. Knowledge Soundness

We will need to apply the following general forking lemma of Bootle et al. [16], [21] to prove knowledge soundness.

Theorem 2. *General forking lemma [16], [21]. Suppose that there exists a polynomial-time witness extractor $\mathcal{E}(\mathsf{crs}, \cdot)$ such that when given any* crs *in the support of* $\mathsf{Gen}(1^\kappa)$ *and a polynomial-size tree of accepting transcripts, $\mathcal{E}$ succeeds with negligible failure in extracting a witness or a non-trivial discrete logarithm relation between the generators in the* crs*.*

Lemma 1. *The protocol presented in Figures 7 satisfies our definition of knowledge soundness (Definition 4) so long as the Discrete Log Assumption (Assumption 6) holds in $\mathcal{G}$.*

Proof: We will show that if $\mathcal{P}$ can succeed in completing the proof for four different challenges from $\mathcal{V}$ for the same initial statement $(\mathbf{g}, h, A)$, $\mathcal{E}$ can either extract a witness $\mathbf{a}$, or find a nontrivial discrete logarithm relation between $\mathbf{g}$ and h. It suffices to show that this holds for one recursive step since the hardness of finding a discrete logarithm relation between $\mathbf{g}'$ and h implies the hardness of computing one between $\mathbf{g}$ and h in Protocol 7. The recursive protocol is the case where the (7a.) branch is ignored in Figure 7.

$\mathcal{E}$ works as follows. Suppose that the four provers produce $\mathbf{a}'_1$, $\mathbf{a}'_2$, $\mathbf{a}'_3$, and $\mathbf{a}'_4$ after respectively receiving z_1, z_2, z_3, and z_4 from the same initial statement such that $z_i \neq z_j$ for $1 \leq i \leq j \leq 4$. We have that for $i \in [4]$:

$$(\mathbf{g}_{[:n']}^{z_i^{-1}} \cdot \mathbf{g}_{[n':]}^{z_i^1})^{\mathbf{a}'_i} \cdot h^{\langle \mathbf{a}'_i, \mathbf{y}_i\rangle} = L^{z_i^2} \cdot A \cdot R^{z_i^{-2}} \tag{3}$$

Now, so long as the following holds

$$\mathbf{det}\begin{bmatrix} z_1^{-2} & z_2^{-2} & z_3^{-2} \\ 1 & 1 & 1 \\ z_1^2 & z_2^2 & z_3^2 \end{bmatrix} \neq 0$$

we can find coefficients v_1, v_2, and v_3 using z_1, z_2 and z_3:

$$\sum_{i=1}^3 v_i z_i^2 = 1, \qquad \sum_{i=1}^3 v_i = 0, \qquad \sum_{i=1}^3 v_i z_i^{-2} = 0.$$

These coefficients yield a discrete logarithm representation of L as follows:

$$\prod_{i=1}^3 ((\mathbf{g}_{[:n']}^{z_i^{-1}} \cdot \mathbf{g}_{[n':]}^{z_i})^{\mathbf{a}'_i} \cdot h^{\langle \mathbf{a}'_i, \mathbf{y}_i\rangle})^{v_i} = \prod_{i=3}^3 (L^{z_i^2} A R^{z_i^{-2}})^{v_i}$$

$$\prod_{i=1}^3 \mathbf{g}_{[:n']}^{v_i z_i^{-1} \mathbf{a}'_i} \cdot \mathbf{g}_{[n':]}^{v_i z_i \mathbf{a}'_i} \cdot h^{v_i \langle \mathbf{a}'_i, \mathbf{y}_i\rangle} = \prod_{i=3}^3 L^{v_i z_i^2} \cdot A^{v_i} \cdot R^{v_i z_i^{-2}}$$

$$\mathbf{g}_{[:n']}^{\sum_{i=1}^3 v_i z_i^{-1} \mathbf{a}'_i} \cdot \mathbf{g}_{[n':]}^{\sum_{i=1}^3 v_i z_i \mathbf{a}'_i} \cdot h^{\sum_{i=1}^3 v_i \langle \mathbf{a}'_i, \mathbf{y}_i\rangle} = L$$

$$\mathbf{g}^{\mathbf{a}_L} \cdot h^{v_L} = L$$

Interactive Inner-Product Argument of Knowledge

- `inner_product_proof`(stmt):
  1. *Setup.* Run $\mathbb{G} \leftarrow \mathcal{G}(1^\kappa)$ and let $g_0, \ldots, g_{n-1}, h$ be random generators of $\mathbb{G}$. Let $\mathbf{g} := (g_0, \ldots, g_{n-1})$ and $\mathsf{crs} := (\mathbb{G}, \mathbf{g}, h)$.
  2. *Input.* Both $\mathcal{P}$ and $\mathcal{V}$ know the statement $\mathsf{stmt} = (A, \mathbf{y}, v)$ where $\mathbf{y} \in \mathbb{Z}_p^n$ and P knows a witness $\mathbf{a}$ such that $v = \langle \mathbf{a}, \mathbf{y} \rangle$ and $\mathbf{g}^{\mathbf{a}} \cdot h^v = A$.
  3. $\mathcal{V}$ challenges $\mathcal{P}$ with a uniform sample $z \xleftarrow{\$} \mathbb{Z}_p^*$.
  4. $\mathcal{P}$ and $\mathcal{V}$ compute $A' := A \cdot h^{z \cdot v}$.
     Return `reduce_proof`$(\mathsf{crs}, A', \mathbf{y}, n; \mathbf{a})$.
- `reduce_proof`$(\mathsf{crs}, A, \mathbf{y}, n; \mathbf{a})$:
  1. if $n = 1$:
     1.1. $\mathcal{P}$ sends $\mathbf{a}$ to $\mathcal{V}$.
     1.2. $\mathcal{V}$ returns the result of $A \stackrel{?}{=} g^a \cdot h^{\langle a, y \rangle}$.
  2. if n is odd:
     2.1. $\mathcal{P}$ sends $\mathbf{a}_{[-1]}$ to $\mathcal{V}$.
     2.2. $\mathcal{P}$ sets $\mathbf{a} := \mathbf{a}_{[:-1]}$.
     2.3. $\mathcal{P}$ and $\mathcal{V}$ update A, $\mathbf{y}$, $\mathbf{g}$, and n as follows:
     $$A := A \cdot \mathbf{g}_{[-1]}^{-\mathbf{a}_{[-1]}} \cdot h^{-(\mathbf{a}_{[-1]} \cdot \mathbf{y}_{[-1]})}$$
     $$\mathbf{y} := \mathbf{y}_{[:-1]}, \quad \mathbf{g} := \mathbf{g}_{[:-1]}, \quad n := n - 1$$
  3. $\mathcal{P}$ computes:
     $$c_L := \langle \mathbf{a}_{[:n']}, \mathbf{y}_{[n':]} \rangle, \qquad c_R := \langle \mathbf{a}_{[n':]}, \mathbf{y}_{[:n']} \rangle$$
     $$L := \mathbf{g}_{[n':]}^{\mathbf{a}_{[:n']}} \cdot h^{c_L}, \qquad R := \mathbf{g}_{[:n']}^{\mathbf{a}_{[n':]}} \cdot h^{c_R}$$
     and sends L and R to $\mathcal{V}$.
  4. $\mathcal{V}$ challenges $\mathcal{P}$ with a uniform sample $z \xleftarrow{\$} \mathbb{Z}_p^*$.
  5. $\mathcal{P}$ and $\mathcal{V}$ both compute:
     $$A' := L^{z^2} A R^{z^{-2}}, \quad \mathbf{g}' := \mathbf{g}_{[:n']}^{z^{-1}} \cdot \mathbf{g}_{[n':]}^{z^1}$$
     $$\mathbf{y}' := z^{-1} \cdot \mathbf{y}_{[:n']} + z^1 \cdot \mathbf{y}_{[n':]}$$
  6. $\mathcal{P}$ computes $\mathbf{a}' := z^1 \cdot \mathbf{a}_{[:n']} + z^{-1} \cdot \mathbf{a}_{[n':]}$.
  7a *Recursive option.* Return:
     `reduce_proof`$((\mathbb{G}, \mathbf{g}', h), A', \mathbf{y}', n'; \mathbf{a}')$.
  7b *Non-recursive option (just for analysis).*
     7b.1. $\mathcal{P}$ sends a' to $\mathcal{V}$.
     7b.2. $\mathcal{V}$ returns the result of the following:
     $\langle \mathbf{a}', \mathbf{y}' \rangle \stackrel{?}{=} c_L \cdot z^2 + c_R \cdot z^{-2} + v \wedge$
     $\mathbf{g}'^{\mathbf{a}'} \cdot h^{c_L \cdot z^2 + c_R \cdot z^{-2} + v} \stackrel{?}{=} A'$.

Fig. 7: Protocol specification of the (non-zero-knowledge) inner-product argument. This protocol is optimized for evaluation proofs by publicizing the $\mathbf{y}$ vector which halves communication complexity with respect to Bulletproofs [21].

where

$$\mathbf{a}_L = (\sum_{i=1}^{3} v_i z_i^{-1} \mathbf{a}_i') \mid\mid (\sum_{i=1}^{3} v_i z_i \mathbf{a}_i') \in \mathbb{Z}_p^n,$$
$$v_L = \sum_{i=1}^{3} v_i \langle \mathbf{a}_i', \mathbf{y}_i \rangle \in \mathbb{Z}_p.$$

By computing fresh coefficients to target the A and R terms respectively, we can obtain discrete logarithm representations of all three terms:

$$L = \mathbf{g}^{\mathbf{a}_L} \cdot h^{v_L}, \quad A = \mathbf{g}^{\mathbf{a}_A} \cdot h^{v_A}, \quad R = \mathbf{g}^{\mathbf{a}_R} \cdot h^{v_R}. \tag{4}$$

Using the equations in (4), we can simplify the equations in (3):

$$\begin{aligned}
&\mathbf{g}_{[:n']}^{z_i^{-1}\mathbf{a}_i'} \cdot \mathbf{g}_{[n':]}^{z_i \mathbf{a}_i'} \cdot h^{\langle \mathbf{a}_i', \mathbf{y}_i \rangle} \\
&\quad = L^{z_i^2} A R^{z_i^{-2}} \\
&\quad = (\mathbf{g}^{\mathbf{a}_L} h^{v_L})^{z_i^2} \cdot (\mathbf{g}^{\mathbf{a}} h^{v_A}) \cdot (\mathbf{g}^{\mathbf{a}_R} h^{v_R})^{z_i^{-2}} \\
&\quad = \mathbf{g}^{\mathbf{a}_L z_i^2 + \mathbf{a}_A + \mathbf{a}_R z_i^{-2}} h^{v_L z_i^2 + v_A + v_R z_i^{-2}} \quad \text{for} \quad i \in [4].
\end{aligned}$$

This implies the following equations for $i \in [4]$:

$$\mathbf{a}_i' z_i^{-1} = \mathbf{a}_{L,[:n']} z_i^2 + \mathbf{a}_{A,[:n']} + \mathbf{a}_{R,[:n']} z_i^{-2} \tag{5}$$
$$\mathbf{a}_i' z_i^1 = \mathbf{a}_{L,[n':]} z_i^2 + \mathbf{a}_{A,[n':]} + \mathbf{a}_{R,[n':]} z_i^{-2} \tag{6}$$
$$\langle \mathbf{a}_i', \mathbf{y}_i \rangle = v_L z_i^2 + v_A + v_R z_i^{-2} \tag{7}$$

Combining equations (5) and (6) above with z_i and z_i^{-1} as the respective coefficients, we have that for $i \in [4]$,

$$\mathbf{a}_{L,[:n']} z_i^3 + (\mathbf{a}_{A,[:n']} - \mathbf{a}_{L,[n':]}) z_i \tag{8}$$
$$+ (\mathbf{a}_{R,[:n']} - \mathbf{a}_{A,[n':]}) z_i^{-1} - \mathbf{a}_{R,[n':]} z_i^{-3} = 0. \tag{9}$$

Now, if the following holds:

$$\det \begin{bmatrix} z_1^{-3} & z_2^{-3} & z_3^{-3} & z_4^{-3} \\ z_1^{-1} & z_2^{-1} & z_3^{-1} & z_4^{-1} \\ z_1^1 & z_2^1 & z_3^1 & z_4^1 \\ z_1^3 & z_2^3 & z_3^3 & z_4^3 \end{bmatrix} \neq 0$$

it must be that:

$$\begin{aligned}
&\mathbf{a}_{L,[:n']} = \mathbf{a}_{R,[n':]} = 0, \\
&\mathbf{a}_{A,[:n']} = \mathbf{a}_{L,[n':]}, \text{ and} \\
&\mathbf{a}_{R,[:n']} = \mathbf{a}_{A,[n':]}.
\end{aligned}$$

We can use these equations to simplify the equations in (9):

$$\mathbf{a}_i' = z_i \mathbf{a}_{A,[:n']} + z_i^{-1} \mathbf{a}_{A,[n':]} \quad \text{for} \quad i \in [4].$$

Thus, $\mathcal{E}$ can extract $\mathbf{a}_A$ knowing z_1, z_2, $\mathbf{a}_1'$, and $\mathbf{a}_2'$ by solving linear equations as long as $z_1^2 \neq z_2^2$. Finally, since equation (7) must hold, it must be that for $i \in [4]$:

$$\begin{aligned}
&c_L z_i^2 + c_R z_i^{-2} + v \\
&= \langle z_i \mathbf{a}_{A,[:n']} + z_i^{-1} \mathbf{a}_{A,[n':]}, z_i^{-1} \mathbf{y}_{[:n']} + z_i \mathbf{y}_{[n':]} \rangle \\
&= z_i^2 \langle \mathbf{a}_{A,[:n']}, \mathbf{y}_{[n':]} \rangle + z_i^{-2} \langle \mathbf{a}_{A,[n':]}, \mathbf{y}_{[:n']} \rangle + \langle \mathbf{a}_A, \mathbf{y} \rangle.
\end{aligned} \tag{10}$$

For equation (10) to hold for z_1, z_2, and z_3, it must be that $\mathbf{a}_A$ satisfies $\langle \mathbf{a}_A, \mathbf{y} \rangle = v$.

■

C. Making the Protocol Zero-Knowledge

The protocol presented in Figures 7 and 7 is *not* zero-knowledge; we present a modified protocol in Figure 8 which achieves perfect zero-knowledge using the non-zero-knowledge inner-product argument as a subroutine.

Zero-Knowledge Argument of Knowledge

inner_product_proof_ZK($\mathbf{s} \in \mathbb{Z}_p^{t+1}, \rho \in \mathbb{Z}_p$):
Setup. Run $\mathbb{G} \leftarrow \mathcal{G}(1^\kappa)$ and let $g, h, g_0, \ldots, g_t$ be random generators of $\mathbb{G}$. Let $\mathbf{g} := (g_0, \ldots, g_t)$.
Input. Both $\mathcal{P}$ and $\mathcal{V}$ know the statement $(A, \mathbf{y}, U)$ where $\mathbf{y} \in \mathbb{Z}_p^n$ and $U \in \mathbb{G}$. $\mathcal{P}$ additionally knows the witness $\mathbf{a} \in \mathbb{Z}_p^{t+1}$ and $r \in \mathbb{Z}_p$ satisfying $\mathbf{g}^{\mathbf{a}} \cdot h^r = A$ and u satisfying $g^u = U$.

1. $\mathcal{P}$ computes $T := g^{\langle \mathbf{s}, \mathbf{y} \rangle}$ and $S := \mathbf{g}^{\mathbf{s}} \cdot h^\rho$, and sends S and T to $\mathcal{V}$.
2. $\mathcal{V}$ challenges $\mathcal{P}$ with a uniform sample $z \in \mathbb{Z}_p^*$.
3. $\mathcal{P}$ computes $\mathbf{c} := \mathbf{a} + z \cdot \mathbf{s}$, $\nu := r + z \cdot \rho$, $\hat{t} := \langle \mathbf{c}, \mathbf{y} \rangle$, and $C := \mathbf{g}^{\mathbf{c}}$. It then sends $\mathbf{c}$, ν, $\hat{t}$, and C to $\mathcal{V}$.
4. $\mathcal{V}$ asserts that the following holds:
   $g^{\hat{t}} \stackrel{?}{=} g^u \cdot T^z \quad \wedge \quad \mathbf{g}^{\mathbf{c}} \cdot h^\nu \stackrel{?}{=} S^z \cdot A \quad \wedge \quad \hat{t} \stackrel{?}{=} \langle \mathbf{c}, \mathbf{y} \rangle$
5. $\mathcal{P}$ and $\mathcal{V}$ interactively compute reduce_proof($(\mathbb{G}, \mathbf{g}, h), A, \mathbf{y}, t+1; \mathbf{a}$).
6. $\mathcal{V}$ outputs success if the inner product proof succeeds.

Fig. 8: Modified wrapper function which first blinds the statement in order to achieve perfect zero-knowledge.

1) Security Proofs: We now show that our modified protocol in Figure 8 satisfies our definitions of security.

Lemma 2. *The protocol presented in Figure 8 satisfies perfect special honest-verifier zero-knowledge (Definition 8).*

Proof: The simulator $\mathcal{S}$ needs to simulate the transcript between $\mathcal{P}$ and $\mathcal{V}$ without knowing the witness. Given the $\mathcal{V}$'s randomness $z \in \mathbb{Z}_p^*$, $\mathcal{S}$ can simulate a protocol transcript for proving an evaluation at point y as follows:

First, the $\mathcal{S}$ chooses $\mathbf{c} \stackrel{\$}{\leftarrow} \mathbb{Z}_p^n$ and $\nu \stackrel{\$}{\leftarrow} \mathbb{Z}_p$ and computes:

$$\hat{t} := \langle \mathbf{c}, \mathbf{y} \rangle, \quad T := \left(\frac{g^{\hat{t}}}{g^u} \right)^{z^{-1}}, \quad S := \left(\frac{\mathbf{g}^{\mathbf{c}} \cdot h^\nu}{A} \right)^{z^{-1}}$$

$\mathcal{S}$ then runs the non-zero-knowledge protocol to simulate the remaining transcript messages. It follows that the simulated transcript are identically distributed to the honestly generated transcript on randomness z. ∎

Lemma 3. *There exists a polynomial-time extractor $\mathcal{E}$ such that given* poly(κ) *accepting transcripts for the same initial statement, $\mathcal{E}$ either extracts a valid witness to the statement or a generalized discrete logarithm relation between the generators $(\mathbf{g}, g, h)$ for the protocol presented in Figure 8.*

Proof: Given poly(κ) accepting transcripts corresponding to the same initial statement at Line 5, we use the extractor of Lemma 1 to extract all witnesses corresponding to the inner-product proof. We proceed if there are at least two.

Given two accepting transcripts with different $(\mathbf{c}_i, z_i, \nu_i, \hat{t}_i)$ tuples for $i \in [2]$, we can compute decommitments for S, A, and T in the exponent as follows:

$$S = (\mathbf{g}^{\mathbf{c}_1 - \mathbf{c}_2} \cdot h^{\nu_1 - \nu_2})^{\frac{1}{z_1 - z_2}},$$
$$A = \mathbf{g}^{\mathbf{c}_1 - \frac{z_1(\mathbf{c}_1 - \mathbf{c}_2)}{z_1 - z_2}} \cdot h^{\nu_1 - \frac{z_1(\nu_1 - \nu_2)}{z_1 - z_2}},$$
$$T = g^{\frac{\hat{t}_1 - \hat{t}_2}{z_1 - z_2}}$$

Thus, we define $\mathbf{s}$, ρ, $\mathbf{a}$, r, and t from the exponents as follows:

$$\mathbf{s} := (\mathbf{c}_1 - \mathbf{c}_2)(z_1 - z_2)^{-1},$$
$$\rho := (\nu_1 - \nu_2)(z_1 - z_2)^{-1},$$
$$\mathbf{a} := \mathbf{c}_1 - z_1(\mathbf{c}_1 - \mathbf{c}_2)(z_1 - z_2)^{-1},$$
$$r := \nu_1 - z_1(\nu_1 - \nu_2)(z_1 - z_2)^{-1},$$
$$t := (\hat{t}_1 - \hat{t}_2)(z_1 - z_2)^{-1}$$

Now, verification equations (Line 4 in Figure 8) must hold for the two accepting transcripts. From $g^{\hat{t}_i} = g^u \cdot h^{\nu_i}$, we get $u = \hat{t}_i - t \cdot z_i$ for $i \in [2]$. Furthermore, from $\mathbf{g}_i^{\mathbf{c}_i} \cdot h^{\nu_i} = S^{z_i} \cdot A$, it must be that $\mathbf{c}_i = \mathbf{s} z_i + \mathbf{a}$ and $\nu_i = \rho z_i + r$ for $i \in [2]$ or we would have a generalized discrete logarithm relation between generators $\mathbf{g}$ and h. Hence if we have not extracted a generalized discrete logarithm relation, we get that $\langle \mathbf{s} z_i + \mathbf{a}, \mathbf{y} \rangle = u + t z_i$ for $i \in [2]$, which implies that $\langle \mathbf{a}, \mathbf{y} \rangle = u$. That is, the extracted components $\mathbf{a}$ and u satisfy the statement being proven. ∎

We can now conclude by proving the following theorem:

Theorem 3. *Assuming that discrete-log relation (Definition 6) holds for the group generator $\mathcal{G}$, the protocol presented in Figure 8 is a secure zero-knowledge argument of knowledge.*

Proof: Perfect completeness follows trivially from the construction. We have perfect special honest-verifier zero-knowledge from Lemma 2. Knowledge soundness follows from Lemma 3 and Theorem 2 of Bootle et al. ∎

Corollary 1. *It follows from the witness-extended emulation lemma of Lindell (Lemma 3.1 of [45]) that the protocol presented in Figure 8 has witness-extended emulation.*

D. Explicit hbPolyCommit Construction

So far, we have presented and proved the security of an interactive inner-product argument protocol. Since the protocol is public-coin, we can convert it into a *non-interactive* protocol that is secure in the random oracle model using the Fiat-Shamir heuristic [12]. For each of the verifier's uniformly random challenges, we instead use random oracle queries which depend on a) the current crs and stmt being proven, and b) the full transcript between the prover and verifier up to the current point. We use the resulting scheme in hbPolyCommit.

Theorem 4. *Assuming that the Discrete Logarithm Relation Assumption (Definition 6) holds for the group generator $\mathcal{G}$, the protocol presented in Figure 9 is a secure polynomial commitment scheme with perfect zero-knowledge.*

Proof: **Correctness.** The verification equations (Step 4 in Figure 8) are easily verified by inspection. Correctness of ProveInnerProduct and VerifyInnerProduct follow from the security of the protocol in Figure 8.

Polynomial Binding. To commit to a polynomial with hbPolyCommit, we commit to the coefficients of the polynomial with a Pedersen vector commitment. Polynomial binding follows from the computational binding property of Pedersen commitments—this holds as long as computing discrete logarithms is intractable in $\mathcal{G}$.

Formal hbPolyCommit Construction

- $\mathsf{Setup}() \to \mathsf{SP}$:
  Let d be the maximum degree for a polynomial that can be committed to with SP. Run $\mathbb{G} \leftarrow \mathcal{G}$, and let $g, h, g_0, g_1, \ldots, g_t$ be random generators of $\mathbb{G}$. Let $\mathbf{g} := (g_1, \ldots, g_d)$. Output $\mathsf{SP} := (\mathbb{G}, g, \mathbf{g}, h)$
- $\mathsf{PolyCommit}(SP, \phi(x)) \to C$:
  Sample $r \xleftarrow{\$} \mathbb{Z}_p$ and let $\mathbf{a}$ denote the coefficient vector for $\phi(x)$. Output $C = (\prod_{i=0}^{deg(\phi)} g_i^{a_i}) \cdot h^r$
- $\mathsf{VerifyPoly}(SP, C, \phi(x), r)$:
  Output $bool := C \stackrel{?}{=} (\prod_{i=0}^{deg(\phi)} g_i^{a_i}) \cdot h^r$
- $\mathsf{CreateEvalProof}(SP, C, i, \phi(i)) \to \pi_i$:
  $\mathbf{y} = (i^0, i^1, \ldots, i^d)$, $\mathbf{s} := (s_0, s_1, \ldots, s_t) \leftarrow \mathbb{Z}_p$,
  $T := g^{\langle \mathbf{s}, \mathbf{y} \rangle}$, $\rho \leftarrow \mathbb{Z}_p$, $S := \mathbf{g}^{\mathbf{s}} h^{\rho}$, $z := \mathcal{H}(SP, C, S, T)$,
  $\mathbf{b} := \mathbf{a} + \mathbf{s}z$, $B := \Pi_{j=0}^{deg(\phi)} g_j^{b_j}$, $\nu := r + \rho z$, $\widehat{t} := \langle \mathbf{b}, \mathbf{y} \rangle$
  Output:
  $\pi_i = (T, S, \widehat{t}, \nu, B, \mathsf{ProveInnerProduct}(SP, \mathbf{b}, \mathbf{y}, \widehat{t}))$
- $\mathsf{VerifyEval}(SP, C, i, \phi(i), \pi_i) \to bool$:
  $z := \mathcal{H}(SP, C, S, T)$
  Output $bool := g^{\widehat{t}} \stackrel{?}{=} g^{\phi(i)} \cdot T^z$ & $B \cdot h^{\nu} \stackrel{?}{=} S^z \cdot C$ & $\mathsf{VerifyInnerProduct}(SP, \mathbf{y}, \widehat{t})$

Fig. 9: Formal specification of hbPolyCommit. The PolyCommit and ProveEval procedures in this scheme are independent since vector commitments are used to commit to the whole polynomial, but a separate argument of knowledge (Figure 8) is used to prove an evaluation.

Strong Evaluation Binding. Suppose that for honestly generated system parameters SP, $\mathcal{A}$ outputs the tuple $(C, \{x_i, y_i, \pi_i\}_{i \in [\ell]})$ for $\ell \geq t+1$, where each π_i is a proof that (x_i, y_i) point lies on the committed polynomial $\phi(\cdot)$, and C is a valid commitment to $\phi(\cdot)$.

For any $(t+1)$-sized subset $S \subseteq [\ell]$, Lagrange interpolation guarantees the extraction of a polynomial consistent with the $t+1$ evaluations corresponding to S. We will prove that regardless of the particular $t+1$-sized subset, Lagrange interpolation will always recover the same polynomial with overwhelming probability.

It follows from knowledge soundness of the embedded proof system (Lemma 1) that there exists polynomial-time extractors $\mathcal{E}_{\mathcal{A},i}$ for $i \in [\ell]$ where the i-th extractor extracts a valid witness of the form $(\phi_i'(\cdot), r_i)$ where $\phi_i'(\cdot)$ is a degree-t polynomial and r_i is the randomness used in the PolyCommit procedure to generate C, and $\phi_i'(x_i) = y_i$.

First, fix the randomness consumed by $\mathcal{A}$ to z (denote this by $\mathcal{A}(z)$). Next, give the honest SP to $\mathcal{A}(z)$ and, in parallel, run the extractors $\mathcal{E}_{A,i}$ for $i \in [\ell]$ on the same SP and randomness z. With overwhelming probability, if $\mathcal{A}$ outputs a polynomial commitment and valid evaluation proofs at ℓ distinct points, then each extractor extracts a witness (ϕ_i', r_i).

Assume there exists a polynomial $\mu(\kappa)$ such that with $1/\mu(\kappa)$ probability, when we generate SP and give it to $\mathcal{A}$, the adversary outputs a polynomial commitment and $t+1$ valid evaluation proofs at distinct points. If this is the case, we say that $\mathcal{A}$ *succeeded*.

Suppose that for some inverse-polynomial probability, the above does not hold for infinitely many κ values. WLOG, assume that $p(\kappa) \geq \kappa^1 0 + 10$. We use the ℓ extractors to construct a polynomial-time algorithm $\mathcal{B}$ that with non-negligible probability finds a commitment with two different openings when given a random SP.

Once a random SP is generated, $\mathcal{B}$ runs $\mathcal{A}$ and waits for its output. $\mathcal{B}$ then runs the ℓ extractors on the same SP and randomness. For each extractor, $\mathcal{B}$ bounds its runtime with a suitably large polynomial $p'(\kappa)$. If any extractor fails to complete in $p'(\kappa)$ time, it outputs *abort*. If all extractors completed their execution, $\mathcal{B}$ checks all of the outputs for a commitment with two openings. WLOG, if $\mathcal{A}$ was not successful, the extractors simply *abort*.

It suffices to show that there exists a suitable choice of $p'(\kappa)$ that depends on $q(\kappa)$ such that with $1 - 2/p^3(\kappa)$ probability, no extractors abort. If true, then the probability that $\mathcal{A}$ is successful, no extractors abort, and the computational binding of the polynomial commitment scheme is not broken is at least $1/p(\kappa) - 2/p^3(\kappa)$. We prove this as follows:

Let $q(\kappa)$ denote $\mathcal{A}$'s maximum runtime such that $\ell \leq q$. Let $p_1(\kappa) = p^3(\kappa) \cdot q(\kappa)$ and p' be a suitable polynomial such that for each extractor, there can be at most $1/p_1(\kappa)$ fraction of bad SP's where at least one extractor exceeds its computational bound $p'(\kappa)$ with probability higher than $1/p_1(\kappa)$. The total probability mass of bad SP's is upper bounded by $\ell/p_1(\kappa) \leq q(\kappa)/p_1(\kappa) = 1/p^3(\kappa)$. Thus, conditioned on a good SP, the probability the extractors do not abort is at least $(1 - 1/p_1(\kappa))^{\ell} \geq (1 - 1/p_1(\kappa))^{q}$. Finally, the probability the extractors do not abort is at least $(1 - 1/p^3(\kappa)) \cdot (1 - 1/p_1(\kappa))^{q(\kappa)} \geq 1 - 2/p^3(\kappa)$.

It follows that since $\ell \geq t+1$, the polynomial interpolated from from the points corresponding to any set $S \subseteq [\ell]$ results in the same polynomial with overwhelming probability.

Perfect Zero-Knowledge. We define the simulator $\mathcal{S} := (\mathcal{S}_1, \mathcal{S}_2)$ as follows:

To simulate the Setup procedure, $\mathcal{S}_1$ calls the underlying zero-knowledge argument of knowledge simulator (guaranteed by Lemma 2) to generate a crs.

$\mathcal{S}_2$ honestly commits to a random univariate polynomial. Whenever the simulator must generate a proof asserting that the committed polynomial evaluate to y at point x, the simulator of the underlying proof system is called to simulate the proof. This is possible because the underlying proof system has PSHVZK (Lemma 2).

We use a simple hybrid argument to prove that the view of $\mathcal{A}$ in the above ideal-world experiment is information-theoretically indistinguishable to its view in the real-world experiment. Essentially, the adversary is given an honest commitment of the polynomial submitted in the real-world experiment rather than the random univariate polynomial. Since Pedersen vector commitments are information-theoretically hiding, the hybrid experiment is identically distributed to the ideal-world experiment. Secondly, since the underlying proof system provides PSHVZK, the hybrid experiment is information-theoretically indistinguishable to the real-world experiment. It follows that any statement passed to the simulator of the underlying inner-product argument must be a true statement. ■

E. hbACSS2 Full Protocol

The primary challenge in realizing hbACSS2 is recreating the effect of Algorithm 2 without access to homomorphic commitments or seperable evaluation proofs. The ideas for solving this are straightforward: 1) the dealer provides the commitments and proofs needed to verify the values that would have been interpolated, 2) the proofs can still be batched along groups that do not need to be separated for share recovery (i.e. each recipient would perform N $\mathsf{BatchVerifyEvals}$). We specify the full hbACSS2 protocol with these changes in Algorithm 3.

1) Notation: When describing hbACSS2, we make use of a few notational conveniences that may not be common in other works. Firstly, we follow every $t+1$ sharing polynomials with $N-(t+1)$ that are interpolated from these first $t+1$ (we call the latter *redundancy polynomials*). By *interpolate* we mean that we use lagrange coefficients to calculate the later polynomials as linear combinations of the former. This is necessary for the dealer to be able to provide proofs that will work for points that would have been interpolated in share recovery.

Another notational convenience we employ is a stepping notation similar to how the $\mathsf{range}()$ function works in Python. For example in line 208, we have $k \in [j, N, P]$ which means to assign k values starting at j and incrementing by N up to and including P.

Algorithm 3 $\mathsf{hbACSS2}(D, \mathcal{P}_1, ..., \mathcal{P}_N)$ for dealer D and parties $\mathcal{P}_1, ..., \mathcal{P}_N$

Setup:

1: Each party begins $\mathcal{P}_i$ with SK_i such that $\mathsf{PK}_i = g^{\mathsf{SK}_i}$
2: The set of all $\{\mathsf{PK}_j\}_{j\in[N]}$ are publicly known
3: Set up the polynomial commitment $\mathsf{SP} \leftarrow \mathsf{Setup}(t)$

As dealer D with input $(s_1, ..., s_B)$:

// Secret Share Encoding

101: Sample B random degree-t polynomials $\phi_1(\cdot)...\phi_B(\cdot)$ such that each $\phi_k(0) = s_k$ and $\phi_k(i)$ is $\mathcal{P}_i$'s share of s_k
102: For every $t+1$ polynomials, interpolate $N-(t+1)$ different degree-t redundancy polynomials $\psi(\cdot)$ for a total of R such polynomials.
103: Let P be the ordered set of polynomials of size $B+R$ such that every consecutive subset of size N contains $t+1$ polynomials from $[B]$ followed by the $N-(t+1)$ redundancy polynomials interpolated from them.

// Polynomial Commitment

104: $\mathbf{C} \leftarrow \{\mathsf{PolyCommit}(SP, \phi_k(\cdot))\}_{k\in[B]}$
105: $\mathbf{C}_r \leftarrow \{\mathsf{PolyCommit}(SP, \psi_k(\cdot))\}_{k\in[R]}$
106: $\mathsf{ReliableBroadcast}(\mathbf{C}, \mathbf{C}_r)$

// Encrypt and Disperse

107: **for** $i \in [1, N]$ **do**
108: **for** $j \in [1, t+1]$ **do**
109: $\pi_{i,j} \leftarrow \mathsf{BatchProveEval}(SP, \mathbf{C}, \{\phi_k(\cdot)\}_{k\in[j,N,P]})$
110: **for** $j \in [t+2, N]$ **do**
111: $\pi_{i,j} \leftarrow \mathsf{BatchProveEval}(SP, \mathbf{C}_r, \{\psi_k(\cdot)\}_{k\in[j,N,P]})$
112: **for** each $\mathcal{P}_i$ **do**
113: $z_i \leftarrow \mathsf{Enc}_{\mathsf{PK}_i}(\{\pi_{i,j}\}_{j\in[1,N]} \| \{\phi_k(i)\}_{k\in[B]})$
114: $\mathsf{Disperse}(\{z_i\}_{i\in[1,N]})$

As receiver $\mathcal{P}_i$:

// Wait for broadcasts

201: Wait to receive $\mathbf{C}, \mathbf{C}_r \leftarrow \mathsf{ReliableBroadcast}$
202: Wait for $\mathsf{Disperse}$ to complete

// Decrypt and validate

203: $z_i \leftarrow \mathsf{Retrieve}(i)$
204: $\{\phi_k(i)\}_{k\in[B]}, \{\pi_{i,j}\}_{j\in[1,N]} \leftarrow \mathsf{Decrypt}_{\mathsf{SK}_i}(z_i)$
205: **if** decryption fails **then** GoTo 212
206: Interpolate $\{\psi_k(i)\}_{k\in[R]}$ from $\{\phi_k(i)\}_{k\in[B]}$
207: **for** $j \in [1, t+1]$ **do**
208: **if** $\mathsf{BatchVerifyEval}(\mathbf{C}, i, \{\phi_k(i)\}_{k\in[j,N,P]}, \pi_{i,j}) \neq 1$ **then**
209: GoTo 212
210: **for** $j \in [t+2, N]$ **do**
211: **if** $\mathsf{BatchVerifyEval}(\mathbf{C}_r, i, \{\psi_k(i)\}_{k\in[j,N,P]}, \pi_{i,j}) \neq 1$ **then**
212: **sendall** $(\texttt{IMPLICATE}, \mathsf{SK}_i)$
213: otherwise, valid shares are owned, so **sendall** `OK`

As receiver $\mathcal{P}_i$ (continued)

// Bracha-style agreement

301: On receiving `OK` from $2t+1$ parties,
302: **sendall** `READY`
303: On receiving `READY` from $t+1$ parties,
304: **sendall** `READY` (if haven't yet)
305: Wait to receive `READY` from $2t+1$ parties,
306: **if** all owned shares are valid (line 213) **then**
307: **output** shares $\{\phi_k(i)\}_{k\in[B]}$

// Handling Implication

401: On receiving $(\texttt{IMPLICATE}, \mathsf{SK}_j)$ from some $\mathcal{P}_j$,
402: ignore if already in *Share Recovery*
403: Discard if $\mathsf{PK}_j \neq g^{\mathsf{SK}_j}$
404: $z_j \leftarrow \mathsf{Retrieve}(j)$
405: **if** one or more checks from lines 205- 211 fail **then**
406: Proceed to *Share Recovery* below

// Share Recovery

Let $\phi(x, y)$ be a degree t, t bivariate polynomial such that $\phi(i, k)$ gives $\mathcal{P}_i$'s share of s_k
Let $\phi_l(x, y)$ be the l'th such polynomial for a total of $L := \lceil B/(t+1) \rceil$ such polynomials
Let $\mathbf{C}_T := \{\mathbf{C}, \mathbf{C}_r\}$
501: **if** we previously received valid shares (line 307) **then**
502: **for** each $\mathcal{P}_j$ **do**
503: **send** $(\texttt{R1}, \{\phi_l(i, j)\}_{l\in[L]}, \pi_{i,j})$ to $\mathcal{P}_j$
504: On receiving $(\texttt{R1}, \{\phi_l(k, i)\}_{l\in[L]}, \pi_{k,i})$ from $t+1$ different parties (with varying values of k) such that $\mathsf{BatchVerifyEval}(\mathbf{C}_T, k, \{\phi_l(k, i)\}_{l\in[L]}, \pi_{k,i}) = 1$,
505: Interpolate $\{\phi(\cdot, i)\}_{l\in[L]}$
506: **for** each $\mathcal{P}_j$ **do**
507: **send** $(\texttt{R2}, \{\phi_l(j, i)\}_{l\in[L]})$ to $\mathcal{P}_j$
508: On receiving $(\texttt{R2}, \{\phi_l(i, k)\}_{l\in[L]})$ from at least $2t+1$ parties,
509: Robustly interpolate $\{\phi_l(i, \cdot)\}_{l\in[L]}$
510: **output** shares $\{\phi_l(i, k)\}_{k\in[t+1], l\in[L]}$