HAVEN++: Batched and Packed Dual-Threshold Asynchronous Complete Secret Sharing with Applications

Nicolas Alhaddad
Boston University
USA
nhaddad@bu.edu

Mayank Varia
Boston University
USA
varia@bu.edu

Ziling Yang
University of Illinois
Urbana-Champaign
USA
zilingy2@illinois.edu

ABSTRACT

Asynchronous complete secret sharing (ACSS) is a foundational primitive in the design of distributed algorithms and cryptosystems that require secrecy. Dual-threshold ACSS permits a dealer to distribute a secret to a collection of n servers so that everyone holds shares of a polynomial containing the dealer's secret.

This work contributes a new ACSS protocol, called HAVEN++, that uses packing and batching to make asymptotic and concrete advances in the design and application of ACSS for large secrets. HAVEN++ allows the dealer to pack multiple secrets in a single sharing phase, and to reconstruct either one or all of them later. For even larger secrets, we contribute a batching technique to amortize the cost of proof generation and verification across multiple invocations of our protocol.

The result is an asymptotic improvement in amortized communication and computation complexity, both for ACSS itself and for its application to asynchronous distributed key generation. We implement HAVEN++ and find that it improves performance over the hbACSS protocol of Yurek et al. by a factor of 3-10× or more across a wide range of parameters for the number of parties and batch size.

KEYWORDS

Distributed systems, Consensus protocols, Asynchronous complete secret sharing, Asynchronous distributed key generation, Polynomial commitments

1 INTRODUCTION

Nearly every cryptographic protocol involving multiple parties begins in the same way: with the the distribution of data related to the parties' cryptographic secrets. This is the first step in group secure messaging to agree upon a shared symmetric key, in threshold cryptosystems to distribute each party's public key material to the group [20], in secure multi-party computation protocols to distribute correlated randomness during preprocessing [1], and more. Rather than broadcasting an (ephemeral) secret from one party to the rest, instead these protocols require each party to disseminate *shares* of their ephemeral secrets that are subsequently used to agree upon common state.

All of these applications can be constructed using an *asynchronous complete secret sharing* protocol, or ACSS. Concretely, ACSS is a distributed protocol that protects a secret that a dealer distributes among n parties. An ACSS protocol has a sharing phase in which a dealer distributes shares of her secret; later, in the reconstruction phase, the parties can collectively recover the secret. After the initial sharing phase is completed, the parties can leverage ACSS' confidentiality, integrity, and availability guarantees to reliably use and compute over the disseminated secrets.

Specifically, an ACSS protocol provides three security guarantees: that all honest parties possess a share of a common secret S after the sharing phase even in the presence of t malicious parties (potentially including the dealer), that the secret S remains confidential if fewer than the threshold number of parties attempt to reconstruct the secret, and that the reconstructed secret x is correct if the dealer is honest. To avoid making brittle assumptions about an upper bound on network latency, ACSS protocols ensure that all three security guarantees hold even if the network reorders or delays messages arbitrarily. Historically, ACSS protocols have been designed to disseminate secrets that are short, e.g., the length of the cryptographic security parameter.

Recent works have considered ACSS to disseminate a large collection of secret material so that threshold cryptosystems can scale to support a larger number of parties (e.g., [42]), messaging or blockchain applications can use each key once and delete it for forward secrecy (e.g., [32, 38]), and MPC correlated randomness (like Beaver triples can be constructed for each gate in a large circuit (e.g., [32, 43]). These works have introduced two efficiency-improving ideas: packing multiple secrets into one ACSS invocation in a somewhat-similar way as packing messages into a homomorphic encryption ciphertext, and performing batched execution of several ACSS sharing phases at once to reduce the field size and enable partial reconstruction.

1.1 Our Contributions

This work makes several asymptotic and concrete advances in the design and application of ACSS with large secrets.

Efficient dual-threshold ACSS protocol. In §3, we contribute a new ACSS construction called HAVEN++. This protocol has optimal amortized communication complexity and rounds, and it avoids the need for a trusted setup or PKI. HAVEN++ is also a *dual-threshold* scheme, which means that it has two different reconstruction protocols with different thresholds: either $t + 1$ parties or $p + 1$ parties are required, for any choice of $p \in (t, n - t)$. HAVEN++ supports reconstruction of all packed secrets under the higher threshold, and reconstruction of a single secret with the smaller threshold.

To improve efficiency on large secrets, we leverage the dual-threshold property in order to *pack* $n - t - p$ secrets inside of a single ACSS invocation. Additionally, we show how to *batch* multiple invocations of HAVEN++ and only disseminate a single combined proof of consistency for the entire batch. This further improves communication complexity, with a significantly lower

Works	word complexity		avoiding setup		number	dual	crypto
	amortized	batch size	no trust?	no PKI?	of rounds	threshold	assumption
Kokoris-Kogias et al. [37]	$O(\kappa n^3)$	$O(\kappa n)$	✓	✗	4	✓	DL
Haven [7]	$O(\kappa n^2)$	$O(1)$	✓	✓	3	✗	DL + RO
e-AVSS [8]	$O(\kappa n^2)$	$O(1)$	✗	✓	3	✗	t-SDH [35]
Das et al. [29]	$O(\kappa n^2)$	$O(1)$	✓	✗	3	✗	DDH + RO
Shoup et al. [41]	$O(\kappa n^2)^\dagger$	$\Omega(n^2)$	✓	✓	7+	✗	RO
hbACSS2 [43]	$O(\kappa n)$	$O(n^2)$	✓	✗	7+	✗	DL + RO
Bingo [1]	$O(\kappa n)$	$O(n)$	✗	✓	7	✓	t-SDH [35]
Haven++	$O(\kappa n)$	$O(n)$	✓	✓	3	✓	DL + RO

Table 1: Our Haven++ ACSS protocol, compared with other AVSS or ACSS protocols. A green background indicates the optimal setting for that cell. Many constructions support batching of sufficiently-large messages, including ours. The table only shows worst-case complexity.

Scheme	Amortized Word Complexity	Batch size	Cryptographic assumption	Setup	Reference
Low threshold ADKG	$O(\kappa n^4)$	1	DDH + RO	PKI	Kate et al. [34]
	$O(\kappa n^3)$	1	DDH + RO	PKI	Das et al. [29]
	$O(\kappa n^3)$	1	SXDH [9] + RO	PKI	Abraham et al. [2]
	$O(\kappa n^2)^\dagger$	$\Omega(n^2)$	RO	None	Groth et al. [32]
	$O(\kappa n)$	$O(n)$	DL + RO	None	This work
High threshold ADKG	$O(\kappa n^4)$	1	DL	None	Kokoris-Kogias et al. [37]
	$O(\kappa n^3)$	1	DL + RO	PKI	Das et al. [28]
	$O(\kappa n^3)$	1	t-SDH [35]	Trusted	Abraham et al. [1]
	$O(\kappa n^2)$	$O(n)$	DL + RO	None	This work

Table 2: ADKG and MPC pre-processing protocols proposed in this paper, compared with the prior state of the art. We show worst-case amortized word complexity, along with the smallest batch size required to reach this amortized cost. The work of Groth et al. [32] includes an optimistic path that achieves $O(n)$ word complexity

minimum requirement on the batch size than prior work. See Table 1 for a detailed comparison between Haven++ and prior works.

Implementation and experimental evaluation. To demonstrate that these techniques improve concrete efficiency in addition to asymptotic efficiency, in §4 we implement our constructions on top of the open-source framework of hbACSS [43], which is the only prior work with optimal (amortized) communication complexity that has an open-source implementation. Our implementation reuses all of their low-level field arithmetic and crypto primitives in order to provide an apples-to-apples comparison.

Our experiments show that Haven++ substantially reduces computation time compared to hbACSS by a factor of at least 3× and often by more than an order of magnitude; the exact savings depends on the number of parties and the batch size (cf. Figures 2-5). We will open source this implementation upon de-identification.

Application to ADKG. Finally, in §5 we contribute a new, non-black-box application of our batched and packed ACSS protocol Haven++ to improve the asymptotic complexity of *asynchronous dynamic key generation*, or ADKG. An ADKG protocol allows a collection of parties to agree on a public key and each possess a share of the corresponding secret key. It is a critical component of distributed protocols like Byzantine agreement [22] and randomness beacons [33], and of cryptographic protocols like threshold signatures [14, 31] and multiparty computation [32, 43].

Our ADKG protocol based on Haven++ has optimal amortized communication complexity, and the amortization "kicks in" at a lower batch size than in prior works. See Table 2 for more details.

1.2 Technical Overview of Our Constructions

In this section, we provide a high-level overview of the main techniques used in our ACSS and ADKG constructions.

ACSS construction, against a DoS adversary. We describe our Haven++ construction here and in Figure 1. For simplicity, we begin by describing the protocol in the (unrealistic) scenario that the t faulty parties will be truthful in any message that they send, i.e., they are only allowed to drop messages.

The Haven++ construction incorporates several design elements of Haven [7] (hence the name), and it also uses a bivariate polynomial ϕ in a related (but not identical) manner as in prior dual-threshold ACSS protocols [1, 37]. Concretely, ϕ is a polynomial of degree p in the horizontal direction and t in the vertical direction. To construct ϕ, the dealer packs $p - t + 1$ secrets on the points of the x-axis to the left of the y-axis; that is, at locations $(-k, 0)$ for $k \in [1, p - t + 1]$. The dealer then randomly chooses sufficiently many points (as shown in the pink shaded region in Fig. 1) to

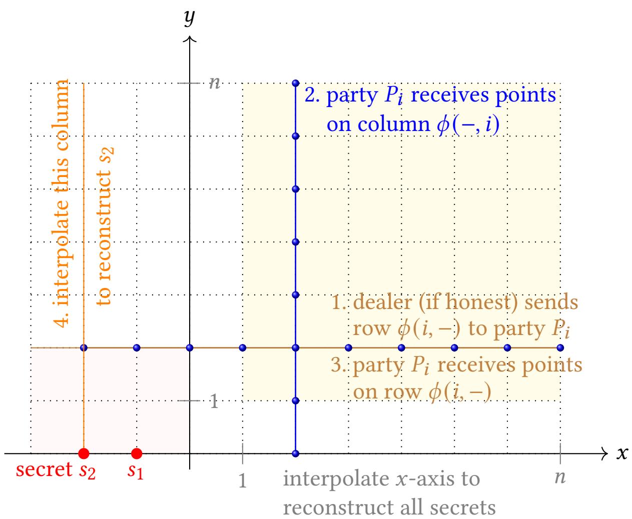

Figure 1: The 2d input space to a bivariate polynomial $\phi(x, y)$ used in our ACSS construction; this example uses $t = 2$, $n = 7$, and $p = 3$. The dealer creates a bivariate polynomial ϕ by packing $p - t + 1 = 2$ secrets s_1 and s_2 on the x-axis (shown in red) and randomly choosing the remaining points in the pink shaded region. In the sharing phase, the parties receive points in the yellow region: specifically, party i learns the row polynomial $\phi(i, -)$ (of degree p, shown in brown) and column polynomial $\phi(-, i)$ (of degree t, shown in blue). Then, the parties can reconstruct all secrets (via the x-axis) or one secret (via the corresponding column, shown in orange).

uniquely determine a $p \times t$ bivariate polynomial. The dealer commits to ϕ in a manner that we specify later, and reliably broadcasts this commitment during the 3 rounds of the sharing phase.

In the sharing phase, each party learns both the i^{th} row and column of ϕ; as a result, the parties collectively learn the points in the yellow shaded region of Fig. 1 (which is disjoint from the pink region). The sharing phase proceeds in three rounds as follows:

(1) The dealer sends each party P_i the i^{th} row polynomial $\phi(i, -)$, as shown in brown in Fig. 1. Looking ahead, the parties will reach agreement if at least $n - 2t$ honest parties receive row polynomials that are consistent with ϕ, but the t malicious parties and up to t honest parties might receive nothing from the dealer.
(2) Any honest party i that received a row polynomial in step 1 now sends the point $\phi(i, j)$ to party j. As a result, *every* honest party receives enough points to interpolate their column polynomial (shown in blue in Fig. 1).
(3) Each party i sends the point $\phi(j, i)$ to party j. After this round, every honest party receives enough points to interpolate their row polynomial, whether or not they received it previously from the dealer in step 1 above.

This construction has $O(\kappa n^2)$ word complexity, where κ denotes both the security parameter and field size of an individual secret. After the sharing phase is complete: $p + 1$ honest parties can reconstruct all secrets by revealing the points on the x-axis, or $t+1$ honest parties can reconstruct a single secret by revealing the points on the corresponding column (shown in orange in Fig. 1).

ACSS construction, against a malicious adversary. To account for malicious parties (including the dealer), we use polynomial and vector commitments so that everyone can prove that they are sending points on the polynomial ϕ. We do not use bivariate polynomial commitments directly in HAVEN++; instead, the dealer produces a polynomial commitment to each *column* polynomial $\hat{\phi}_i = \text{pCom}(\phi(-, i))$ and broadcasts a vector $\vec{v} = [\hat{\phi}_i]_{i \in [n]}$ of these commitments. Anyone who receives $\vec{v}$ is eventually assured that these polynomial commitments collectively correspond to a bivariate polynomial because: (i) they can check directly that each commitment corresponds to some column polynomial of degree at most t, and (ii) collectively they know that at least $t + 1$ parties have checked that $t + 1$ rows are of degree p and by lemma $B.1$ this vector commitment must have been for a bivariate polynomial commitment.

Whenever the parties send a point $\phi(i, j)$, they always attach a proof of inclusion on the corresponding column polynomial ϕ_j. We emphasize that even in steps 1 and 3 that involve sending points on a row, the corresponding proofs are still with respect to the respective column polynomials instead.

These proofs become the dominant cost in the HAVEN++ protocol. For this reason, the protocol benefits even more by batching $O(n)$ executions in parallel, which collectively contain a total of $O(n^2)$ secrets. We construct a batch proof in Algorithm 4 that efficiently proves correctness of all $O(n)$ row or column polynomials at once. Batching improves the amortized communication complexity by a factor of n, as shown in Table 1.

ADKG construction. Asynchronous dynamic key generation allows n parties to agree on a public key such that each party holds one share of the corresponding secret key. In this work, we construct an ADKG protocol by combining n instances of ACSS (one per party) with a multi-valued Byzantine agreement or MVBA protocol. The full construction is shown in Figure 6 and Algorithm 6; we provide only a brief overview here.

The high-level idea of our construction is simple: each of the n parties acts as the dealer and disperses a bivariate polynomial, and then we "mix and match" columns from everyone to form $t + 1$ new bivariate polynomials that the adversary does not fully know.

One challenge here is that some parties might act maliciously as the dealer and disperse shares that will never reach agreement. This is where the multi-valued Byzantine agreement protocol comes in: it allows the n parties to 'vote' on which $n - t$ of the dispersed bivariate polynomials to use in the mix-and-match stage. Confidentiality is maintained even if the adversary knows t of these polynomials.

Another challenge is that ADKG requires polynomial arithmetic "in the exponent," since it is used to determine a public/secret key pair in a group where the discrete logarithm is assumed to be hard. Fortunately, all of our polynomial operations like evaluation and interpolation are linear, so they can be performed in the exponent. See §5 for details.

1.3 Related Work

AVSS and ACSS. The problem of asynchronous verifiable secret sharing, or AVSS, dates back to at least the 1990s. Early works (e.g., [12, 23, 24]) showed the feasibility of AVSS with unconditional security (i.e., without any cryptographic assumptions), but they had a large communication complexity. In the early 2000s, Cachin et al. [20] made two important advances: designing an AVSS with optimal message complexity against $t < n/3$ malicious parties, and a dual-threshold AVSS where correctness holds against $t < n/4$ parties yet secrecy holds against $t < n/2$ parties. However, both constructions suffer from suboptimal $O(\kappa n^3)$ communication complexity.

The last few years have seen a renaissance of work in this field, with several works that improve asymptotic and concrete performance, reduce computational assumptions and the need for trusted setup, and increase the thresholds for correctness and secrecy. Many recent works (though not all) leverage recent innovations in the design of polynomial commitments, vector commitments, and succinct zero-knowledge proofs. Kokoris-Kogias et al. [37] and Alhaddad et al. [7] constructed "high-threshold" AVSS protocols that maintain secrecy for up to $p < 2n/3$ parties. Backes et al. [8] constructed the first AVSS with $O(\kappa n^2)$ communication complexity, and Das et al. [29] and Shoup et al. [41] improve concrete efficiency. The recent hbACSS protocol of Yurek et al. [43] and Bingo protocol of Abraham et al. [1] are achieve amortized linear complexity. These recent works also emphasize the need for a *complete* secret sharing (i.e., ACSS rather than AVSS) such that the parties can reconstruct the entire polynomial rather than just the secret, which has applications in the MPC setting.

ADKG. Asynchronous distributed key generation enables robust, fault-tolerant communication over an unreliable network. As such, it is a valuable building block toward many distributed protocols, including those used for threshold cryptography and blockchains/state machine replication. Several of the works cited above also consider ADKG, as do standalone works like Das et al. [28].

Polynomial and vector commitments. Our construction uses polynomial and vector commitments as a building block to achieve consensus on the bivariate polynomial ϕ. We consider two polynomial commitment schemes: KZG commitments [36] that have constant-sized proofs but require trusted setup, and Bulletproofs [18] that have transparent setup but require log-sized proofs. Our Haven++ protocol uses these commitments in a black-box manner, so we could alternatively use any polynomial commitment scheme that is deterministic and homomorphic (e.g., [15, 16, 19]). We also use the related idea of vector commitments, which were initially introduced by Libert-Yung [39] and Catalano-Fiore [26].

2 PRELIMINARIES

In this section, we introduce our model and some of building blocks that are going to be used in this paper for our constructions:

2.1 Model

We study a network of n parties, each pair interconnected via an authenticated and private channel. A malicious adversary, denoted by $\mathcal{A}$, can corrupt up to t parties. Our network is asynchronous: $\mathcal{A}$ can delay but must eventually deliver messages between honest parties.

2.2 Definitions and Building Blocks

2.2.1 *Dual-threshold Verifiable Secret Sharing.*

Secret sharing is a method where a secret is divided into shares in such a way that only specific subsets of shares can reconstruct the original secret. Verifiable Secret Sharing (VSS) enhances this by allowing participants to verify that their shares reconstruct to the same secret, even in the presence of a bad dealer. In verifiable secret sharing, an attacker is allowed to control t out of n parties, and if the attacker doesn't corrupt the dealer then the attacker learns nothing about the secret being shared. Moreover, any $t + 1$ parties can reconstruct the secret.

Dual-threshold verifiable secret sharing adds another degree of flexibility: the reconstruction threshold p is not restricted to $t + 1$ but can be higher. Dual-threshold asynchronous verifiable secret sharing has two phases: the sharing phase and the reconstruction phase. In the sharing phase, a special party called the *dealer* disperses a secret s among the n parties. The following definition is adapted from Abraham et al. [1] to incorporate the dual threshold property.

Definition 2.1 (AVSS [1]). A dual-threshold asynchronous verifiable secret sharing protocol contains three protocols `Share`, `Reconstruct`, and `Reconstruct`(k) that satisfy the following three properties, even against an adversary who controls t malicious parties.

- *Termination*: If the dealer is honest, then all honest parties will complete `Share`. Also if one honest party completes `Share`, then all honest parties will. Finally, if all honest parties complete Share and invoke `Reconstruct` or `Reconstruct`(k), then they all will complete reconstruction.
- *Correctness*: All honest parties who complete the partial reconstruction protocol `Reconstruct`(k) should agree on the same secret. The same is true for `Reconstruct`, and moreover it should produce the same secret at location k. Finally, this secret should be the same as the one initially used by the dealer in the `Share` protocol, if the dealer was honest.
- *Secrecy*: An adversary should not be able to learn anything about the k^{th} secret until the point at which some honest party invokes `Reconstruct`(k). For the full reconstruction protocol `Reconstruct`, an adversary should not be able to learn anything even if it participates in the protocol with up to $p - t$ honest parties.

An asynchronous *complete* secret sharing protocol, or ACSS, additionally satisfies the completeness property

Definition 2.2 (Completeness [29]). If some honest party completes `Share`, then there exists a degree-t polynomial p such that $p(0) = s$ and each honest party i will eventually hold a share $s_i = p(i)$. Moreover, when the dealer is honest, s is the secret that it initially shared.

2.2.2 Reed-Solomon Error Correcting Code. Reed-Solomon error-correcting codes play a fundamental role in state-of-the-art reliable broadcast protocols, verifiable secret sharing schemes, and information-theoretic multi-party computation protocols. In adversarial settings, they empower honest parties to reconstruct the dealer's secret (or the plaintext message in reliable broadcasts) even amidst failures.

Formally, an (m, n) error-correcting code is defined by a pair of algorithms ($\mathsf{ECCEnc}, \mathsf{ECCDec}$). The encoding algorithm, denoted by $\mathsf{ECCEnc}(M, m, k)$, ingests a message M comprised of k symbols, interprets it as a polynomial of degree $k-1$, and emits m evaluations of said polynomial. Conversely, the decoding algorithm, represented by $\mathsf{ECCDec}(k, r, T)$, receives a set of symbols T—some potentially erroneous—and produces a polynomial of degree $k-1$, or equivalently, k symbols. This is achieved by amending up to r errors (incorrect symbols) within T. It is a well-established fact [40] that ECCDec can rectify up to r errors in T and yield the initial message given that $|T| \geq k + 2r$.

Note that in this paper we will only formally call the decoding algorithm. The encoding algorithm will not be called.

2.2.3 Online Error Correcting. Online error correction (OEC) refers to a set of techniques where error detection and correction are performed as data is transmitted or processed. It was first used by Canetti et al. [13] for doing verifiable secret sharing and was later used for asynchronous reliable broadcast [29], [4] and asynchronous verifiable information dispersal [5]. In contrast to traditional error-correcting codes that first gather all data before starting the correction procedure, online methods operate as data streams in. This capability is particularly useful to honest parties that are trying to filter out bad shares as they are receiving them. For our use case, when $n = 3t + 1$ and when all honest parties have evaluation of a polynomial of degree t, it allows a receiver to recover the polynomial of degree t after hearing from $2t + 1$ honest parties, even though t parties might send bad evaluations. We refer the reader to the original paper of Canetti et al. [13] for full details.

2.3 Polynomial and Vector Commitments

We consider polynomial commitment scheme that allows a prover to commit non interactively to a polynomial such that, later, the prover can be asked to open the commitment at any particular point and reveal the corresponding value. We follow the same definition as HAVEN; like them, we require the polynomial commitment to be determinsitic and additively homormorphic. We re-state the definition for convenience with the addition of three extra optional algorithms DoubleBatchProof, BatchVerify and BatchProof that are all used strictly for the batched variant of our algorithm described in §3.2.

Definition 2.3. A *polynomial commitment scheme* $\mathcal{P}$ comprises four algorithms Setup, pCom, Eval, Verify and four optional algorithms Hom, DoubleBatchProof, BatchVerify, BatchProof that act as follows:

- $\text{Setup}(1^\kappa, \mathbb{F}, D) \rightarrow \text{pp}$ is given a security parameter κ, a finite field $\mathbb{F}$, and an upper bound D on the degree of any polynomial to be committed. It generates public parameters pp that are required for all subsequent operations.
- $\text{pCom}(\text{pp}, \phi(x), d) \rightarrow \hat{\phi}$ is given a polynomial $\phi(x) \in \mathbb{F}[x]$ of degree $d \leq D$. It outputs a commitment string $\hat{\phi}$ (throughout this work, we use the hat notation to denote a commitment to a polynomial).
- $\text{Eval}(\text{pp}, \phi, i) \rightarrow \langle i, \phi(i), w \rangle$ is given a polynomial ϕ as well as an index $i \in \mathbb{F}$. It outputs a 3-tuple containing i, the evaluation $\phi(i)$, and witness string w_i.
- $\text{Verify}(\text{pp}, \hat{\phi}, y, d) \rightarrow$ True/False takes as input a commitment $\hat{\phi}$, a 3-tuple $y = \langle i, j, w \rangle$, and a degree d. It outputs a Boolean.
- $\text{Hom}(\text{pp}, \hat{\phi}_1, \hat{\phi}_2, a) \rightarrow \widehat{\phi_1 + a\phi_2}$ takes in commitments to two polynomials ϕ_1 and ϕ_2 of degree at most D, as well as a field element $a \in \mathbb{F}$. Outputs the commitment $\text{pCom}(\text{pp}, \phi, \max\{d_1, d_2\})$ to the polynomial $\phi = \phi_1 + a\phi_2$.
- $\text{BatchProof}(\text{pp}, \phi, n', d)$ is given a polynomial ϕ of degree d, where n' is a positive integer. It outputs n' 3-tuples, each containing i where $0 < i < n' + 1$, the evaluations $\phi(i) \ldots \phi(n')$, and n proofs $w_1 \ldots w_n$.
- $\text{DoubleBatchProof}(\text{pp}, [\phi_1, \ldots, \phi_n], n', d)$ is given a list of polynomials $\phi_1 \ldots \phi_n$ of the same degree d, where n is a positive integer, as well as a positive integer n'. It outputs n 3-tuples, each containing i where $0 < i < n' + 1$, the evaluations $\phi_1(i) \ldots \phi_n(i)$, and one batch proof witness string w_i.
- $\text{BatchVerify}(\text{pp}, [\hat{\phi}_1, \ldots \hat{\phi}_n], y, d)$ takes as input a list of polynomial commitments $[\hat{\phi}_1, \ldots \hat{\phi}_n]$, a 3-tuple $y = \langle i, j_1 \ldots j_n, w \rangle$, and a degree d. It outputs True, if $\phi_z(i) = j_z \; \forall z, 0 < z < n{+}1$ and False otherwise.

We also use vector commitments in this work; these are also succinct commitments to a large set of data, but the data need not correspond to points on a polynomial. That is, vector commitments are cryptographic primitives that allow one to commit to an ordered sequence of values (or a vector) and later prove the value of a specific position in the vector without revealing any other information about the rest of the vector. Much like polynomial commitments, the commitment size is constant, not dependent on the size of the vector. For our implementation, we instantiate HAVEN++ with Merkle trees for vector commitments.

Due to space limitations, we defer to Appendix A a formal definition of vector commitments and the security guarantees that both styles of commitments must satisfy.

2.4 Private Polynomial Commitments

For our ADKG construction in §5.2, we require an additional property of our polynomial commitment. A prover must demonstrate that a value is a correct polynomial evaluation in the exponent, while maintaining the privacy of the polynomial evaluation itself. The verifier, upon receiving the polynomial commitment, the index, the evaluation in the exponent, and the proof, can return true if the claim is correct or false otherwise.

(1) $\mathit{PrivateEval}(\text{pp}, \phi, i) \rightarrow \langle i, g^{\phi(i)}, w \rangle$ is given a polynomial ϕ as well as an index $i \in \mathbb{F}$. It outputs a 3-tuple containing i, the evaluation in the exponent $g^{\phi}(i)$, and witness string w_i. Here g is a random generator of a group G where discrete log is hard.

(2) $PrivateVerify(\text{pp}, \hat{\phi}, y, d) \rightarrow$ True/False takes as input a commitment $\hat{\phi}$, a 3-tuple $y = \langle i, j, w \rangle$, and a degree d. It outputs True if the verification succeeds, and False otherwise.

We instantiate private polynomial commitments on top of any additively homomorphic polynomial commitment in Appendix C.

2.5 Distributed algorithms

In this section, we provide a formal definition of multi-valued Byzantine agreement (MVBA). Looking ahead, we use this primitive as a building block in §5 to construct an asynchronous distributed key generation (ADKG) protocol.

MVBA was initially introduced by Cachin et al. [21]. As the name suggests, it generalizes Byzantine agreement to allow for a message that is more than one bit in length. Additionally, Abraham et al. [3] introduced the ability for parties to check that the agreed-upon message satisfies some predicate Q, possibly when matched with some additional information w that is also decided during the protocol. A precise definition follows.

Definition 2.4 (MVBA). A multi-valued Byzantine agreement scheme is an interactive protocol between n parties that satisfies the following five criteria.

- *Termination*: If all messages between honest parties have been delivered, then all honest parties will terminate with an output v.
- *Agreement*: If an honest party outputs v, then every honest party also terminates and outputs v.
- *External validity*: Every honest party that terminates decides v validated by w such that the predicate $Q(v, w)$ is true.
- *Integrity*: If all parties follow the protocol, and if an honest party decides v validated by w, then some party proposed v validated by w.
- *Quality*: The probability of terminating with a value v that was proposed by a correct replica is at least 1/2.

3 OUR ACSS CONSTRUCTION

In this section, we introduce our dual-threshold ACSS protocol, called Haven++ that achieves all the properties of a dual-threshold ACSS as shown in Appendix B. We present the construction in two parts: first with packing of multiple secrets into a single bivariate polynomial, and then batching across multiple bivariate polynomials. For simplicity and without loss of generality we instantiate our protocol with $n = 3t + 1$ (optimal resilience) and with $p = 2t$.

3.1 Haven++ with Packing

Haven++ has two phases: a sharing phase in which the dealer distributes shares of her secret s, and a reconstruction phase in which the parties collectively reconstruct one or more secrets.

Sharing phase. The construction of Haven++ is heavily influenced by Haven [7]. It operates in three rounds of communication, and follows the same communication pattern as Bracha's asynchronous reliable broadcast [17]. However, unlike Haven, our construction uses a bivariate polynomial when producing the shares and uses a distributed check to make sure that recovery polynomials are consistent with the shares.

Below, we describe the protocol for the optimal resilience case of $n = 3t + 1$. Conceptually, the protocol contains three distinct steps.

(1) The broadcast step (lines 1-8): The dealer samples a random bivariate polynomial $\phi(X, Y)$ such that each row polynomial is of degree $2t$ and each column polynomial is of degree t. Also, the row polynomial at index 0, encodes b secrets. Each packed secret $s_k \in s_1 \dots s_b$, is packed at $\phi(-k, 0)$. Using pCom, the dealer commits to the first n column polynomial of degree t (lines 3-5). The dealer evaluates each column polynomial $\phi_i(Y)$ at indices $1 \dots n$ and at the same time produces n proofs for every point on every column by calling Eval (lines 6-7). Remember that Eval returns both the (x, y) coordinate as well as a proof that this coordinate is on the committed polynomial. The dealer then sends each row (not column) of evaluation proofs with all n polynomial commitments to every party (lines 6-8).

(2) The echo phase (lines 9-14): when a party i receives the first broadcast message from the dealer: The party verifies that the row evaluation proofs are consistent with the column polynomial polynomial commitments and checks that the evaluation points are on a polynomial of degree $2t$ (line 10). If both check pass, then the party commits using a vector commitment to all the polynomial commitments (line 11) in the the same order it got from the dealer and produces proofs of inclusion (lines 12-13). It then then sends an echo message to every party p_j containing the vector commitment C, the polynomial commitment $\hat{\phi}_j$, the corresponding inclusion proof π_j and the evaluation proof at j, $\vec{y}_i[j]$ (lines 12-14). Note that when party p_i sends an echo message, she doesn't yet know whether her polynomial commitments will become the consensus ones, because the Bracha broadcast protocol on C might not be complete

(3) The ready phase (lines 15-28): When a party p_i hears $2t + 1$ echo messages from different parties with the same vector commitment C, the proper column polynomial commitment and its inclusion proof at position i and with $2t + 1$ valid evaluation proofs on the column polynomial. p_i interpolates its own column polynomial and generates n evaluation proofs (line 18). p_i then sends each party j the j^{th} evaluation on its own column polynomial (every point on the column of this party, is also a point on the row of another party) with its own polynomial commitment, inclusion proof and C (line 19).
To guarantee that every honest party sends a ready, just like Bracha broadcast protocol, we also have an amplification step. If a party didn't send a ready and hears $t + 1$ ready. The party waits until it hears $t + 1$ valid echo messages instead of $2t + 1$. This condition must be met eventually because at least one honest party heard $2t+1$ echo messages where $t + 1$ must have came from honest parties. Once $t + 1$ valid echo messages are heard the protocol continues as before. i.e. p_i interpolates its own column polynomial and generates n evaluation proofs (line 24). p_i then sends each party j the $j^t h$ evaluation on its own column polynomial with its own polynomial commitment, inclusion proof and C (line 25).

Algorithm 1 Sharing phase of HAVEN++, for server P_i and tag ID.d.

1: UPON RECEIVING (ID.d, in, share, $s_1 \dots s_b$): ▷ only if party is the dealer P_d
2: Uniformly sample $\phi(X, Y)$ with degree $2t$ in X and t in Y such that $\phi(-k, 0) = s_k, \forall k \in [0, b]$.
3: **for** $i \in [1, n]$ **do**
4: $\phi_i(Y) = \phi(i, Y)$
5: $\hat{\phi}_i = \text{pCom}(\text{pp}, \phi_i(Y), t)$ ▷ Commit to every column polynomial
6: **for** $i \in [1, n]$ **do**
7: compute $\vec{y}_i = [\text{Eval}(\text{pp}, \phi_j(Y), i)$ for $j \in [1, n]]$ ▷ evaluate and create witnesses for every point on every column polynomial
8: send "ID.d, send, set$_i$" to party P_i, where set$_i = \{[\hat{\phi}_1 \cdots \hat{\phi}_n], \vec{y}_i\}$ ▷ send every party i all column polynomial commitments and the i^{th} evaluation of every column polynomial with the proper opening proof

9: UPON RECEIVING (ID.d, send, set$_i$) from P_d for the first time: ▷ echo stage
10: **if** $\forall j$, Verify(pp, $\phi_j, \vec{y}_i[j], t$) and all points $(j, \vec{y}_i[j])$ form a degree $2t$ polynomial **then**
11: $C = \text{vCom}(\bar{\text{pp}}, [\hat{\phi}_1 \cdots \hat{\phi}_n])$ ▷ commit to all polynomial commitments
12: **for** $j \in [1, n]$ **do**
13: $\pi_j = \text{vGen}(\bar{\text{pp}}, \hat{\phi}_j, j)$ ▷ send message to each party P_j
14: send "ID.d, echo, info$_{i,j}$" to P_j, where info$_{i,j} = \{C, \hat{\phi}_j, \pi_j, \vec{y}_i[j]\}$

15: UPON RECEIVING (ID.d, echo, info$_{m,i}$) from P_m for the first time: ▷ ready stage
16: **if** vVerify(pp, $C, \hat{\phi}_i, i, \pi_i$) and Verify(pp, $\hat{\phi}_i, \vec{y}_i[m], t$) = True **then**
17: **if** not yet sent ready and received $2t + 1$ valid echo with the same C **then**
18: interpolate $\phi_i = \phi(i, Y)$ from any $t + 1$ valid $\vec{y}_i[m]$ in the received echo ▷ interpolate column i with the help of other honest parties
19: send "ID.d, row, info$_{i,j}$" to P_j where info$_{i,j} = (y_i = \text{Eval}(\text{pp}, \phi_i, j), \hat{\phi}_i, \pi_i, C)$ ▷ completes Bracha consensus on C, and also sends to party j a point on their row

20: UPON RECEIVING (ID.d, ready, info$_{m,i}$) from P_m for the first time:
21: **if** vVerify(pp, $C, \hat{\phi}_m, m, \pi_m$) and Verify(pp, $\hat{\phi}_m, \vec{y}_m[i], t$) = True **then**
22: **if** not yet sent ready and received $t + 1$ valid ready with this C **then**
23: wait to receive $t + 1$ valid echo with this C ▷ must happen eventually
24: interpolate $\phi_i = \phi(i, Y)$ from any $t + 1$ valid $\vec{y}_i[m]$ ▷ interpolate column i with the help of other honest parties
25: send "ID.d, row, info$_{i,j}$" to P_j where info$_{i,j} = (y_i = \text{Eval}(\text{pp}, \phi_i, j), \hat{\phi}_i, \pi_i, C)$ ▷ Bracha consensus on C, while also sending to party j, a point on their row
26: **if** received $2t + 1$ valid ready messages **then**
27: interpolate $\phi(X, i)$ from the $2t + 1$ valid ready messages ▷ construct the row polynomial from the column points of other parties
28: output (ID.d, out, shared) ▷ locally halt

Algorithm 2 Reconstruction phase of HAVEN++ for all packed secrets, for server P_i and tag ID.d

1: UPON RECEIVING (ID.d, in, reconstruct) from P_i for the first time:
2: send (ID.d, reconstruct-share, $\hat{\phi}_m, y^*_m$) to all parties ▷ from Party P_m

3: UPON RECEIVING (ID.d, reconstruct-share, $\hat{\phi}_m, y^*_m$): ▷ from Party P_m
4: **if** $\hat{\phi}_m$ in C and Verify(pp, $\hat{\phi}_m, y^*_m, t$) **then**
5: **if** received $2t + 1$ valid reconstruct-share messages **then**
6: interpolate $\phi(X, 0)$ from the $2t + 1$ valid points
7: output (ID.d, out, reconstructed, $\phi(-k, 0) = s_k, \forall k \in [0, b]$)

When a party hears $2t + 1$ ready messages from every different party p_m with the same vector commitment C, the proper column polynomial commitment ϕ_m and its inclusion proof π_m at position m and with $2t+1$ valid evaluation proofs for $\phi_m(i)$ (line 26). Party interpolates its row polynomial and finishes the dispersal.

Reconstruction phase. We provide two algorithms for reconstruction; the first one (Algorithm 2) enables reconstruction of all secrets at the same time (or one secret that requires a high threshold to

Algorithm 3 Reconstruction phase of Haven++ for a packed secret share j, for server P_i and tag ID.d

Upon receiving (ID.d, in, reconstruct, j) from P_i for the first time:
    send (ID.d, reconstruct-share, $\phi(-j, m)$) to all parties ▷ from Party P_m

Upon receiving (ID.d, reconstruct-share, $\phi(-j, m)$): ▷ from Party P_m
    if received at least $t + 1$ reconstruct-share **then** ▷ Run Online Error correcting code
        $\tilde{\phi_{-j}}$ = ECC(points, t, e) ▷ with e initialized to 1, attempt to interpolate the column polynomial at $-j$, $\phi(-j, Y)$
        if $\tilde{\phi_{-j}} \neq \perp$ **then**
            output (ID.d, out, reconstructed, $\phi_{-j}(0)$)
        e = e + 1 ▷ increase the number of errors by one with each failed decoding

recover), and the second one (Algorithm 3) allows selective opening of a specific secret. Both protocols are simple: each party sends one or more points on the bivariate polynomial to the recipient along with corresponding proofs. The most important difference between the two protocols is *which* points are included. As shown in Fig. 1, reconstructing all secrets requires interpolation on a polynomial of degree p, whereas reconstructing a single secret requires interpolation on a different polynomial of degree t. Both types of reconstruction are possible because each party holds one row and one column of data after the sharing phase.

Theorem 3.1. *Assuming that the underlying polynomial and vector commitment schemes satisfy Definitions A.2-A.5, then the Haven++ protocol is a dual-threshold ACSS with $O(\kappa n^2 c)$ communication complexity, where κ is the security parameter and c is the size of the underlying polynomial commitment and evaluation.*

We provide in Appendix B, the proofs for each of the properties in Definition 2.1.

3.2 Haven++ with Batching

In this section we explain how we can batch multiple invocation of Haven++ to save on practical amortized communication complexity and at the same time enhance run time. The core idea is instead of the dealer generating one packed bivariate polynomial, the dealer has to generate a batch of them. Haven++ can then batch across many bivariate polynomials. However, for efficient batching, we need to introduce polynomial batching across multiple polynomials.

3.2.1 Batching for multiple polynomial evaluation. Consider a set of n polynomial commitments $\{\hat{\phi}_1 \dots \hat{\phi}_n\}$ and a single evaluation point j. If a prover wants to prove that $\phi_i(j) = y_i$ for all $i \in \{1 \dots n\}$, then the prover has to send n witnesses, one for each $(y_i, \hat{\phi}_i)$. However, one can build one succinct proof for all n polynomials at index j.

To achieve this, hbACSS [43] built their own "Batch Inner-Product" that generalizes the inner product argument of Bulletproofs [43] and used it to empower their ACSS construction. Similarly Alhaddad et al. [6], built a generic construction that works for any additive homomorphic polynomial commitments. For this work, we instantiate the generic construction of Alhaddad et al. [6] using both Bulletproofs [18] and AMT [42] with some modifications that we describe below.

To recall, the scheme of Alhadadd et al. [6] is a three-round sigma protocol that is made non-interactive using the Fiat-Shamir transform. It has three steps: (1) a commitment, (2) a public coin challenge, and (3) a response.

(1) Commitment: prover commits to n different polynomials $\{\phi_1 \dots \phi_n\}$ with the same degree d. For each $i \in \{1, \dots, n\}$, the prover runs $\hat{\phi}_i$ = pCom(pp, ϕ_i, d) and sends the pair $(\phi_i(j), \hat{\phi}_i)$.
(2) Challenge: verifier generates a random point $c \in \mathbb{F}$
(3) Response: prover interpolates ϕ_c from $\{\phi_1 \dots \phi_n\}$ and sends the witness w after running Eval(pp, $\hat{\phi}_c$, j)

The verifier computes $\phi_c(j)$ by interpolating all $\phi_i(j)$, where $i \in \{1, \dots, n\}$. Subsequently, the verifier calculates $\hat{\phi}_c = \sum_{i=1}^{n} \hat{\phi}_i \cdot l_i(c)$, where l_i represents the Lagrange basis polynomial. The verifier can compute $\hat{\phi}_c$ thanks to the additive homomorphic property of the employed polynomial commitment scheme. The verifier accepts the proof if Verify(pp, $\hat{\phi}_c$, $(j, \phi_c(j), w)$, d) returns True, and rejects it otherwise.

Consider a prover who wishes to perform n simultaneous evaluation proofs for a verifier. If implemented naively with Fiat-Shamir, the prover has to generate n different challenges and interpolate n different polynomials. Instead, we adopt the same approach used in hbACSS, where we employ a common challenge for all proofs across all verifiers.

If a prover intends to create multiple proofs across various indices, it is not possible to send every set of evaluations to each verifier individually. To address this, we construct a Merkle tree in which each leaf node contains all of the transcripts associated with a specific verifier, and the root hash of the Merkle tree serves as the shared challenge. Subsequently, we provide each verifier with a Merkle branch, allowing them to reconstruct the root hash and verify that it fully encompasses all of the verifier's transcripts. We formally define our batching in algortithm 4.

3.2.2 Construction. To support efficient batched calls to Haven++, our batched Haven++ protocol makes some changes to the sharing phase in Algorithm 1. The main changes can be summarized with the following:

(1) the dealer creates n' batches of bivariate polynomoials each packing b secrets.
(2) The dealer calls DoubleBatchProof to generate n proofs for b univariate column polynomials at a time (instead of $n * b$ proofs).
(3) Instead of every party vector committing to all the univariate polynomial commitment directly. They commit to lists of column polynomial commitments (of size n') where

Algorithm 4 Double Batch Proof Algorithm

Require: *PolynomialScheme*, a polynomial commitment scheme is that is additively homomorphic.

DOUBLEBATCHPROOF(pp, $[\phi_1, \ldots, \phi_n], n', d$)
  Initialize an empty list W
  for $i \in [1, n']$ **do**
    $leaf_i = H(PolynomialScheme.\text{pp}, i, [\phi_1(i) \ldots \phi_n(i)])$
  Let $root$ be the Merkle tree root and let $branch_i$ be the Merkle branch of $leaf_i$.
  Interpolate ϕ_{root} from $[(1, \phi_1), \ldots, (n, \phi_n)]$.
  $w_1 \ldots w_{n'} = \text{BatchProof}(\text{pp}, \phi_{root}, n', d)$
  append w_i to the list W
  return W
BATCHVERIFY(pp, $[\hat{\phi}_1, \ldots \hat{\phi}_n], y, d$)
  $\langle i, [\phi_1(i), \ldots, \phi_n(i)], branch_i \mid\mid w_i \rangle = y$
  $leaf_i = H(PolynomialScheme.\text{pp}, i, [\phi_1(i) \ldots \phi_n(i)])$
  if not Merkltree.verify($branch_i$, $leaf_i$) **then**
    return *False*
  Let $root = branch_i[0]$ ▷ get the root of the Merkle tree from the Merkle proof
  Let $\hat{\phi}_{root}$ be the interpolation of $[(1, \hat{\phi}_1), \ldots (n, \hat{\phi}_n)]$ at root ▷ This is possible because the polynomial commitment scheme is additively homomorphic
  return Verify(pp, $\hat{\phi}_{root}, w_i, d$)

each column polynomial commitment is from a different bivariate polynomial.

(4) Every party can verify n' points at time (each point is on a column that belong to a different bivariate polynomial) using BatchVerify.

We now offer more details into the full protocol:

- Broadcast phase: Instead of generating one bivariate polynomial, the dealer generates a batch n' of bivariate polynomials. Each bivariate polynomial packs b secrets $s_1 \ldots s_b$ (line 2). To cater for a list of bivariate polynomials, the dealer commits using pCom to n univariate polynomials of every bivariate polynomial (lines 3-5). Instead of producing proofs for every evaluation point, we now batch across all n' bivariate polynomials using the algorithm DoubleBatchProof 4. In more details, for a specific column j, $\phi_{i,j}$ of every bivariate polynomial $i \in [0, n']$ is used as input to the new algorithm DoubleBatchProof, which will produce n proofs i.e DoubleBatchProof(pp, $[\phi_{1,j} \ldots \phi_{n,j}], n, t$). Party p_i will receive the i^{th} proof. Instead of sending one row to every party, the dealer sends n' rows (i^{th} row of every bivariate polynomial) at a time with n proofs (see Fig. 7), regardless of how big n' can be. Still, the dealer has to send all $(n * n')$ polynomial commitments and polynomial evaluations to every party.
- The echo phase (lines 9-14): Each party p_i verifies that all row evaluations and proofs are consistent with the column polynomials. One batch of n' univariate column polynomial at a time. Let C_j be the j^{th} univariate polynomial commitment of every bivariate added together in a list and let $r_1 \ldots r'_n$ be the row evaluation of every bivariate polynomial. The verifier calls BatchVerify($C_j, \langle j, [r_1[j] \ldots r'_n[j]], \vec{y}_i[j] \rangle, t$)). This allows the verifier check n' points each from every row at the same time, for one proof. Also, just like before, each row of evaluations must be on a degree $2t$ polynomial (line 10).
  Instead of committing to all univariate polynomial commitments directly (line 11), party p_i vector commits to every C_i (after hashing it) instead, producing one vector commitment C. Remember that every C_i is a list of column polynomial commitments, where each polynomial commitment is at index i of the list of batched bivariate polynomials. Just as before the party also produces proofs of inclusion with the exception that that every element inside of the vector commitment is a hash to a list of polynomial commitments instead of hash of a single polynomial commitment. (lines 12-13) The party then sends an echo message to every party p_j containing the vector commitment C, the polynomial commitments C_j, the corresponding inclusion proof π_j and the evaluation proof at j, $\vec{y}_i[j]$ (lines 12-14).
- The ready phase (lines 15-28): When a party p_i hears $2t + 1$ echo messages from different parties with the same vector commitment C, the proper column polynomial commitments and its inclusion proof at position i and with $2t + 1$ valid evaluation proofs for all column polynomials. p_i interpolates its own n' column polynomials and generates n evaluation proofs by calling DoubleBatchProof on the list of column polynomials it just interpolated. p_i then sends each party j the $j^t h$ evaluation on its own column polynomial for every bivariate polynomial with its own polynomial commitments C_i, inclusion proof and C (line 19).
  If a party didn't send a ready and hears $t + 1$ ready. The party waits until it hears $t + 1$ valid echo messages instead of $2t + 1$. Once $t + 1$ valid echo messages are heard the protocol continues as before. i.e. p_i interpolates its own n' column polynomials and generates n evaluation proofs by calling DoubleBatchProof (line 24). p_i then sends each

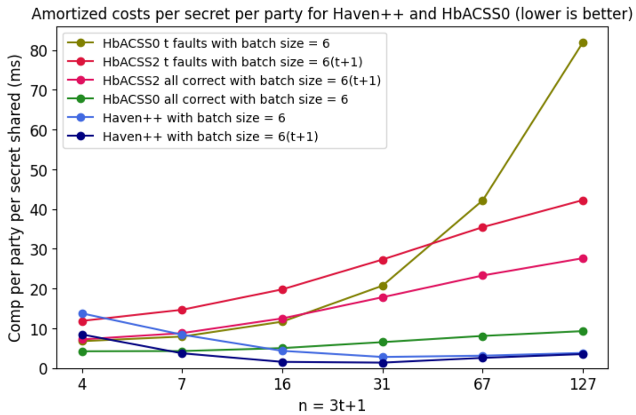

Figure 2: The computational cost per party for HAVEN++, compared against hbACSS0 and hbACSS2 from Yurek et al. [43] (lower is better). For sufficiently many parties n, HAVEN++ wins by a factor of $3\times$ or more, for any batch size.

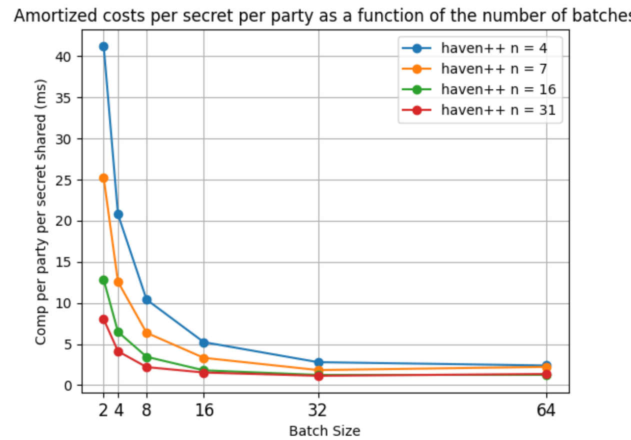

Figure 3: The computational cost per party in HAVEN++ is reduced as the number of batches increases. Each batch packs t+1 secrets. (Each curve has a fixed number of parties n, and lower is better.)

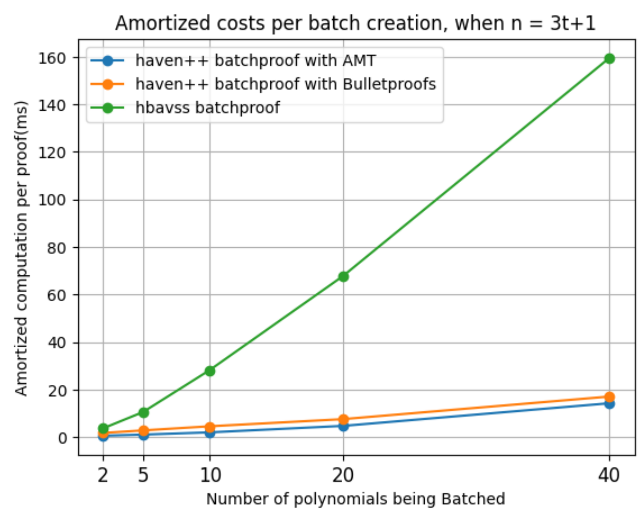

Figure 4: HAVEN++ batching instantiated with AMT [42] and Bulletproofs [18] vs hbACSS batch proof [43]. Our batch proof substantially beats hbACSS in batch creation for any number of polynomials batched.

party j the j^{th} evaluation on its own column polynomial for every bivariate polynomial with its own polynomial commitments C_i, inclusion proof and C (line 25).

When a party hears $2t + 1$ ready messages from every different party p_m with the same vector commitment C, the proper column polynomial commitments C_m and its inclusion proof π_m at position m and with $2t+1$ valid evaluation proofs (line 26). The party interpolates n' row polynomials and finishes the dispersal.

4 EXPERIMENTAL RESULTS

We have implemented our protocol in Python, and we will make our code open-source upon de-anonymization. In this section, we describe some features of our code, and we show detailed experimental results and comparisons to the hbACSS family of protocols.

Implementation details. The implemented version of the HAVEN++ protocol generally performs the same computation as shown in Algorithms 1-3 and described in §3. However, there are a few noteworthy differences that we describe below.

First, in the dealing step, the dealer sends both a row polynomial and column polynomial to every party, rather than just a row polynomial. This has no impact on confidentiality or asymptotic communication, and provides a small efficiency boost in the honest dealer setting. This is because if a party p_i has received a column and row polynomial from the dealer for a specific root commitment C, and other parties are sending echos and ready messages with evaluations with the root commitment C, then p_i can disregard verifying and using those evaluations because p_i already has that information from the dealer.

Second, the implemented version uses fast Fourier transforms over the roots of unity in order to evaluate and interpolate the polynomials efficiently. The implemented version also uses batching to batch multiple bivariate polynomials as described in §3.2, and it is also a multi-core implementation that delegates each bivariate polynomial to a different core.

Experimental setup. We implemented our AVSS protocol using Python 3.7.3. on top of the existing open source implementation of hbACSS [43]. For easy comparison, we use the same Python wrapper that implements the elliptic curve (BLS12-381) in Rust (field size is 381 bits), the same setting of $n = 3t + 1$, and also run in the same docker container as hbACSS for easy reproduciblity of both our work and theirs. The containers run on our machine which has an an AMD 3700X CPU (released in 2019) with 8 main cores and 80GB RAM. Note that our AVSS comfortably runs with less than 8GB of RAM.

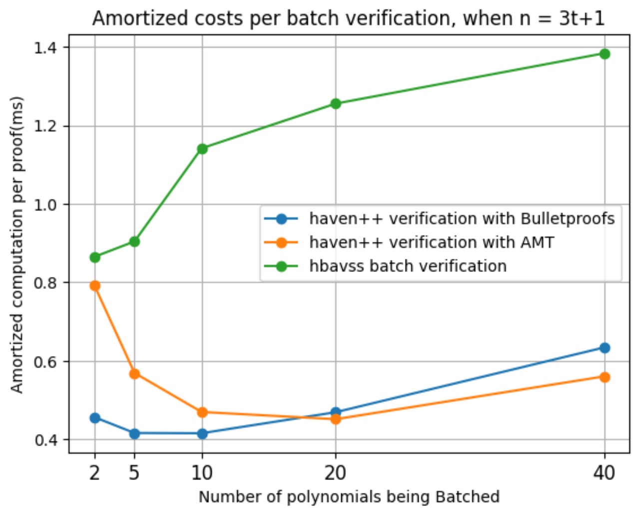

Figure 5: Haven++ batching instantiated with AMT [42] and Bulletproofs [18] vs hbACSS's batch proof verification [43]. Our batch proof verification substantially beats hbACSS in batch creation for any number of polynomials batched.

We use asyncio for managing concurrent communication. All parties are simulated using a single core and run one after the other in a queue. It is important to note that our dealer makes use of multiple cores when generating the proofs to simulate a real use case scenario and exploit the way we batch proofs. Also, parties use multiple cores to verify the dealer message. However, when verifying messages from each other no multi-core is used.

The experimental results shown in Yurek et al. [43] are extrapolated: they run their protocol with dummy polynomial commitments, and use this to estimate the total runtime. By contrast, the figures shown in this work are based on actual executions of their protocol using the primitives that they have developed.

Experimental results. We show the results of our experiments in Figures 2-5. The overall results are that our construction scales better to a large message with a large number of parties, using less CPU and RAM resources and benefiting from batching.

Figure 2 compares the performance of Haven++ and the two fastest variations of hbACSS, namely variants 0 and 2. Variant 0, is the fastest in the optimistic case where there are no faults, while Variant 2 is better in the pessimistic case where there are t faults. We show results for a constant batch size, and for a batch size that scales linearly in the number of parties. The figure shows that as n scales, our amortized computational cost per party remains low whereas hbACSS grows rapidly.

The remaining figures show the impact of batching. Figure 3 shows how Haven++ becomes more efficient as the batch size grows, even while holding fixed the number of parties and the number of secrets packed into each polynomial. Figures 4 and 5 show the costs of creating and verifying a proof, respectively. They show a 2-8× speedup for Haven++ relative to hbACSS.

5 APPLICATIONS OF ACSS

ACSS is a building block for a wide array of applications, including Byzantine Agreement, Weak Leader Election (and Weak Common Coin), ADKG and Asynchronous Multi Party Computation in general. Our ACSS serves as a drop in replacement, for many of the AVSS and ACSS primitives in those applications.

We will provide a brief overview of each application before focusing primarily on our ADKG application. This focus is twofold: (1) to introduce a novel method of combining randomness, which could be of independent interest, and (2) to highlight new asymptotic improvements in amortized word complexity.

(1) **Byzantine Agreement and Weak Leader Election**: Haven++ shares the same interface with Bingo: (1) it allows a dealer to share up to k secrets where $k \leq t+1$, (2) it allows reconstruction of any single secret if $t+1$ parties are present, and (3) it allows to reconstruct the sum of those secrets. As such, Haven++ can also be used to create Validated Byzantine Agreement and weak common coin in the same way that Bingo did, i.e., by using the techniques of Abraham et al. [2] of running gather and of Canetti et al. [25] to generate weak common coin from verifiable secret sharing (note that this implies weak leader election).
(2) **Asynchronous MPC**: To make MPC practical, the use of pre-processing in MPC has been instrumental. Parties agree on shared correlated randomness in the offline phase, and can then utilize them subsequently in the online phase to speed up their computation. A prominent example of such a technique is the generation of Beaver triples [10], where parties maintain secret shares of three variables a, b, and c with the condition that $c = ab$. Although, we conjecture that Haven++ could enhance the amortized communication complexity for generating these triples, we made the observation that Haven++ is also capable of generating equally potent MPC pre-processing materials. Specifically, Haven++ enables the generation of dual threshold shares $t, 2t$ of common random secrets, enabling multiplication in the online phase with just one round of communication ($b = 1$) [11]. For further details, we direct the reader to Appendix D. When the distributed randomness technique of §5.1 is applied, Haven++ enables the generation of the pre-processing material with a word complexity of just $O(n^2\lambda)$ without the need for a trusted dealer with $b = 1$ and $p = 2t$, and $n = 3t + 1$. To the best of our knowledge, this efficiency represents an improvement by a factor of $O(n)$ compared to similar pre-processing methods like beaver triples.
(3) **ADKG**: Haven++ can be used to produce both low and high threhsold ADKG; we provide constructions of both in §5.2. Our ADKGs have on amortized word complexity that is $O(n)$ better than Bingo [1] and without using trusted setup. To do this, we introduce a new way of generating randomness from bivariate polynomials (discussed in §5.1) and use it as a stepping stone for our ADKG construction described in §5.2. For simplicity, we assume that we have a black box access to an Multi-Valued Byzantine Agreement (MVBA) protocol, rather than building one from Haven++.

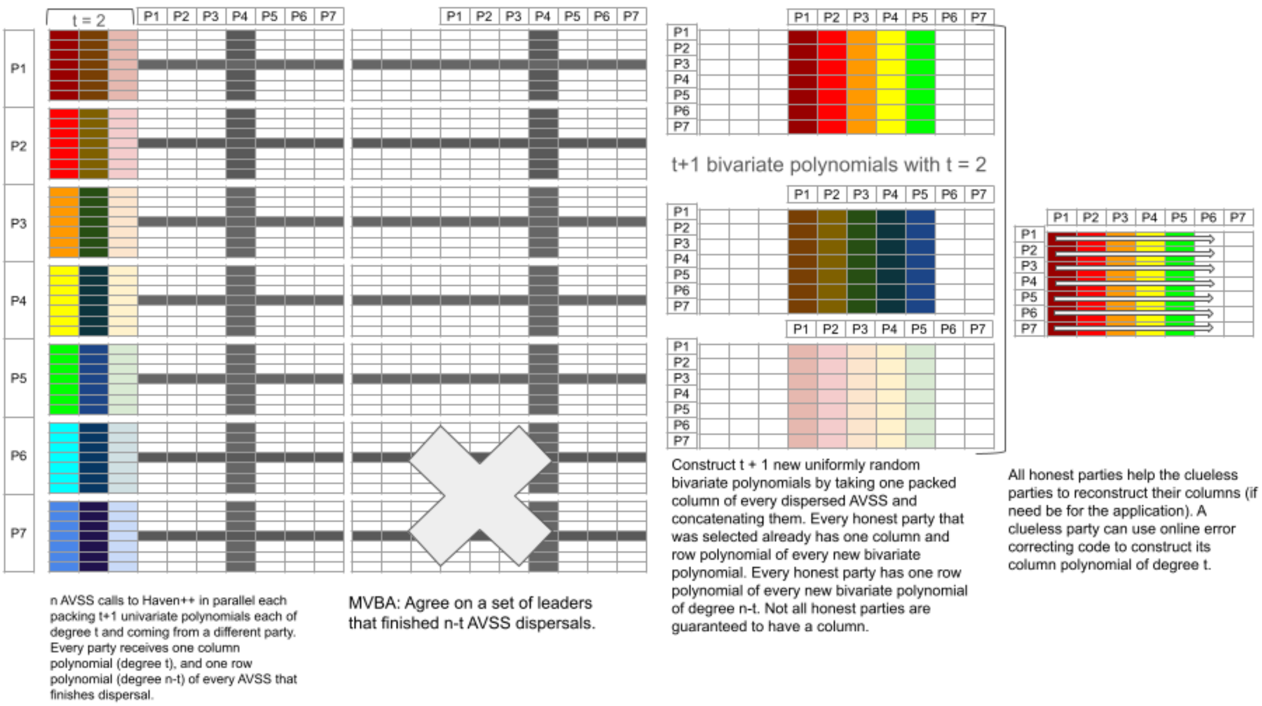

Figure 6: Producing $t + 1$ uniform random bivariate polynomials from Haven++. In this figure, $n = 3t + 1$ where $t = 2$.

5.1 Generating Distributed Secret Shared Randomness

A common technique to generate distributed secret shared common randomness among n parties is to have every party secret-share a random secret using ACSS. An MVBA protocol (as defined in §2.5) is run by the parties to agree on a set of $n - t$ (sometimes $t+1$) parties that finished the sharing phase. Afterward, every party sums up the shares from the set of parties that finished dispersal and reveals the sum of the shares. The common random secret is computed as the sum of all those secrets. The idea is that as long as one party is honest and chooses its own secret uniformly at random then the output is random. Notice, that this technique requires to sacrifice n calls to the ACSS disperse to get one common random number. In this section, we show how to use n calls to Haven++ (with $b = t + 1$) to produce $O(n^2)$ distributed random secrets that are secret shared using low threshold or $O(n)$ distributed random secrets that are secret shared using high threshold ($b = 1$).

We summarize our Distributed Randomness protocol in Algorithm 6 and describe it below:

5.1.1 Sharing Phase: Each party p_i samples b uniformly random numbers from a finite field $s_1 \dots s_b$. Each party p_i then calls the dispersal phase of Haven++ and passes the secrets as input.

5.1.2 Agreement Phase: During the agreement phase, parties run a multi-valued validated Byzantine agreement (MVBA) protocol to reach consensus on a subset of terminated dispersals of Haven++. Specifically, each party i waits for a set S_i of $2t + 1$ instances to finish. Party i subsequently inputs S_i into the MVBA protocol. Additionally, party p_i maintains a comprehensive set S representing all instances that have terminated so far that is incrementally updated. For a given value S_j provided to the MVBA by party p_j, party p_i leverages the predicate $P(S_j, S)$ to verify that $|S_j| \geq 2t + 1$ and $S_j \subseteq S$, confirming that all dispersal instances within S_j have indeed terminated at party p_i. Upon completion, the MVBA protocol yields a set T, where $|T| \geq 2t + 1$. Following the determination of set T by the MVBA protocol, every party p_i pick the first $2t + 1$ parties of the set T and remove the rest.

5.1.3 Randomness Extraction Phase: For every party p_j that was in the set T, party p_i saves the row polynomial that it got from the dispersal phase of that party and all the column polynomial commitments. The first part is to set the column polynomial commitments for each new bivariate polynomial. To this end p_i has to first compute homomorphically the column polynomials that has been shared by every p_j. Let $\hat{\phi}_{-b} \dots \hat{\phi}_0$ be the polynomial commitments for party p_j that has been computed by p_i. Party p_i considers each $\hat{\phi}_k$ as the j^{th} polynomial commitment of the $-k$ bivariate polynomial where, $-b \leq k \leq 0$. Now that p_i has polynomial commitments for every new bivariate polynomial, p_i has to compute its row polynomials. Party p_i first evaluates the row polynomial row_j (acquired from Haven++'s dispersal of p_j) at b points from $[-b, 0]$. Recall that at those locations, each evaluation, would be a share of the packed secret with threshold t (as shown in Fig 6). Party p_i considers each $(1, row_1(-k)), (2, row_2(-k)) \dots (2t+1, row_{2t+1}(-k))$ as row_i of the $-k$ bivariate polynomial where, $-b \le k \le 0$.

5.1.4 *Distribution Phase:*

To guarantee that every party has columns and not just rows, party p_i sends every party p_j, $r_1(j) \dots r_b(j)$ evaluations where $r_1 \dots r_b$ are the row polynomials for the newly distributed random bivariates (as shown in Figure 6). If a party doesn't have some columns (because it wasn't picked as part of the MVBA set), then that party uses online error correcting code to reconstruct its columns from other parties.

5.2 Low and High Threshold Distributed Key Generation (Extracting Public Keys)

In this section, we show how the protocol from section 5.1 can be extended, with the help of private polynomial commitments, to build either a low, or high threshold distributed key generation protocol.

To recall, at the end of our distributed randomness protocol in Algorithm 6, every party has a row and column of every bivariate polynomial. In other words, for every bivariate polynomial, every party p_i has $t+1$ secret shares of $t+1$ different secrets (each of degree t), or one secret share of one secret that is of degree $2t$. Moreover, every party has a univariate polynomial commitment for every column of every bivariate polynomial. What's left is to have public keys that correspond to packed univariate polynomial commitment(s). To this end, we use private polynomial commitments as defined in §2.4.

Let g be a random generator of a group G where discrete log is hard. We show how to get public keys for one distributed bivariate polynomial but without lost of generality the same can be done with all bivariates. Let ϕ_i be the column polynomial that party p_i has. p_i runs $PrivateEval(\text{pp}, \phi_i, 0)$ to generate the tuple $y_i = \langle i, g^{\phi_i(0)}, w \rangle$ and sends it to all parties. Every party j verifies that the tuple (and the public key is correct) by checking that $PrivateVerify(\text{pp}, \hat{\phi}_i, y_i, d)$ returns True. Recall that Party j already has consensus over all polynomial commitments (party i doesn't have to send it). Once party p_i hears $2t+1$ valid private polynomial commitments from $2t+1$ different parties. For every packed secret k, where $-b \le k \le 0$, party p_i computes $g^k = \prod g^{\phi_j(0) L_j(k)}$ where j is the index of the party that sent a valid private polynomial commitment and L is the Lagrange polynomial.

In case of high threshold, $b = 0$, there will only be one public key per bivariate polynomial. While in case of low threshold, $b = t+1$, every bivariate would be packing $t+1$ public keys that can be evaluated from $z = -b$ to $z = 0$.

5.2.1 *Word Complexity Analysis.*

Our ADKG construction can be decomposed into the sum of the costs for our ACSS, MVBA, and the transmission of private polynomial commitment evaluations. The total cost is as follows:

- HAVEN++ incurs a word complexity of $O(\kappa n^2)$ when instantiated with Bulletproofs. In our ADKG, we have n calls to our HAVEN++, which results in a word complexity of $O(\kappa n^3)$.
- The MVBA incurs a word complexity of at most $O(\kappa n^3)$, if instantiated with the latest MVBAs ([30], [1], [2]).
- Broadcasting the private polynomial commitments incurs a word complexity of $O((\kappa n)$ per party per bivariate polynomial. We have a batch of $O(n)$ $(t+1)$ polynomials and we have n parties. Thus a total word complexity of $O((\kappa n^3)$

For high threshold ADKG, it is possible to construct $O(n)$ $(t+1)$ ADKGs (when $b = 1$), with the total amortized word complexity being $O(\kappa n^2)$. For low threshold ADKG, it is possible to construct $O(n^2)$ $((t+1) \times (t+1))$ ADKGs with $b = t+1$. The total amortized word complexity would thus be $O(\kappa n)$.

6 CONCLUSION

This work presents a dual-threshold ACSS construction called HAVEN++, which doesn't require trusted setup and is free from any complaint phase. HAVEN++ accommodates both low and high thresholds to reconstruct the entire message, and it boasts an optimal number of rounds and communication complexity while also demonstrating practical efficiency. Furthermore, we introduce an innovative method to construct an ADKG (Asynchronous Distributed Key Generation) system by utilizing secret shared bivariate polynomials coupled with private polynomial commitments.

ACKNOWLEDGMENTS

This material is based on work supported by DARPA under Agreement No. HR00112020021 and by the National Science Foundation under Grants No. 1801564, 1915763, 2209194, 2217770, and 2228610.

REFERENCES

[1] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, and Gilad Stern. Bingo: Adaptivity and asynchrony in verifiable secret sharing and distributed key generation. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, pages 39–70, Cham, 2023. Springer Nature Switzerland.
[2] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. *Distributed Computing*, 36(3):219–252, Sep 2023.
[3] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In Peter Robinson and Faith Ellen, editors, *38th ACM PODC*, pages 337–346. ACM, July / August 2019.
[4] Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Balanced byzantine reliable broadcast with near-optimal communication and improved computation. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, page 399–417, New York, NY, USA, 2022. Association for Computing Machinery.
[5] Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Brief announcement: Asynchronous verifiable information dispersal with near-optimal communication. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, page 418–420, New York, NY, USA, 2022. Association for Computing Machinery.
[6] Nicolas Alhaddad, Sisi Duan, Mayank Varia, and Haibin Zhang. Succinct erasure coding proof systems. Cryptology ePrint Archive, Paper 2021/1500, 2021. https://eprint.iacr.org/2021/1500.
[7] Nicolas Alhaddad, Mayank Varia, and Haibin Zhang. High-threshold AVSS with optimal communication complexity. In Nikita Borisov and Claudia Díaz, editors, *FC 2021, Part II*, volume 12675 of *LNCS*, pages 479–498. Springer, Heidelberg, March 2021.
[8] Michael Backes, Amit Datta, and Aniket Kate. Asynchronous computational VSS with reduced communication complexity. In Ed Dawson, editor, *CT-RSA 2013*, volume 7779 of *LNCS*, pages 259–276. Springer, Heidelberg, February / March 2013.
[9] Lucas Ballard, Matthew Green, Breno de Medeiros, and Fabian Monrose. Correlation-resistant storage via keyword-searchable encryption. Cryptology ePrint Archive, Paper 2005/417, 2005. https://eprint.iacr.org/2005/417.
[10] Donald Beaver. Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, 4:75–122, 1991.
[11] Zuzana Beerliová-Trubíniová and Martin Hirt. Simple and efficient perfectly-secure asynchronous mpc. In Kaoru Kurosawa, editor, *Advances in Cryptology*

– *ASIACRYPT 2007*, pages 376–392, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
[12] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *25th ACM STOC*, pages 52–61. ACM Press, May 1993.
[13] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *STOC*, pages 52–61. ACM, 1993.
[14] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 31–46. Springer, Heidelberg, January 2003.
[15] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 327–357. Springer, Heidelberg, May 2016.
[16] Jonathan Bootle and Jens Groth. Efficient batch zero-knowledge arguments for low degree polynomials. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018, Part II*, volume 10770 of *LNCS*, pages 561–588. Springer, Heidelberg, March 2018.
[17] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.
[18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018.
[19] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 677–706. Springer, Heidelberg, May 2020.
[20] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In Vijayalakshmi Atluri, editor, *ACM CCS 2002*, pages 88–97. ACM Press, November 2002.
[21] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 524–541. Springer, Heidelberg, August 2001.
[22] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, July 2005.
[23] Ran Canetti. *Studies in secure multiparty computation and applications*. PhD thesis, Citeseer, 1996.
[24] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *25th ACM STOC*, pages 42–51. ACM Press, May 1993.
[25] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 42–51, New York, NY, USA, 1993. Association for Computing Machinery.
[26] Dario Catalano and Dario Fiore. Vector commitments and their applications. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 55–72. Springer, Heidelberg, February / March 2013.
[27] Alessandro Chiesa, Michael A. Forbes, and Nicholas Spooner. A zero knowledge sumcheck and its applications. *Electron. Colloquium Comput. Complex.*, 24:57, 2017.
[28] Sourav Das, Zhuolun Xiang, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous high-threshold distributed key generation and distributed polynomial sampling. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5359–5376, Anaheim, CA, August 2023. USENIX Association.
[29] Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2705–2721. ACM Press, November 2021.
[30] Sisi Duan, Xin Wang, and Haibin Zhang. Fin: Practical signature-free asynchronous common subset in constant time. Cryptology ePrint Archive, Paper 2023/154, 2023. https://eprint.iacr.org/2023/154.
[31] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20(1):51–83, January 2007.
[32] Jens Groth and Victor Shoup. Fast batched asynchronous distributed key generation. Cryptology ePrint Archive, Paper 2023/1175, 2023. https://eprint.iacr.org/2023/1175.
[33] Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018.
[34] Aniket Kate, Yizhou Huang, and Ian Goldberg. Distributed key generation in the wild. Cryptology ePrint Archive, Report 2012/377, 2012. https://eprint.iacr.org/2012/377.
[35] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2010.
[36] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, December 2010.
[37] Eleftherios Kokoris-Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1751–1767. ACM Press, November 2020.
[38] Chelsea Komlo and Ian Goldberg. FROST: Flexible round-optimized Schnorr threshold signatures. In Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn, editors, *SAC 2020*, volume 12804 of *LNCS*, pages 34–65. Springer, Heidelberg, October 2020.
[39] Benoît Libert and Moti Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 499–517. Springer, Heidelberg, February 2010.
[40] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*, volume 16. Elsevier, 1977.
[41] Victor Shoup and Nigel P. Smart. Lightweight asynchronous verifiable secret sharing with optimal resilience. Cryptology ePrint Archive, Paper 2023/536, 2023. https://eprint.iacr.org/2023/536.
[42] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan-Gueta, and Srinivas Devadas. Towards scalable threshold cryptosystems. In *2020 IEEE Symposium on Security and Privacy*, pages 877–893. IEEE Computer Society Press, May 2020.
[43] Thomas Yurek, Licheng Luo, Jaiden Fairoze, Aniket Kate, and Andrew Miller. hbacss: How to robustly share many secrets. In *NDSS*. The Internet Society, 2022.

A DEFINING VECTOR AND POLYNOMIAL COMMITMENTS, CONTINUED

In this section, we provide a precise specification of the methods in a vector commitment scheme. Then, we rigorously define the security guarantees for both vector and polynomial commitments.

Defining vector commitments. This paper follows the convention of Alhaddad et al. [7] and restricts attention to commitment schemes that are *deterministic* and *homomorphic*. We leverage the deterministic property in the HAVEN++ construction so that vector and polynomial commitments do not need a separate "decommitment randomness" string that itself would need to be reliably dispersed.

This approach does introduce one security challenge: a deterministic commitment cannot be hiding when used to commit to an arbitrary secret, only for a randomly-chosen secret. This challenge is solvable through a simple blinding technique previously used by [7, 15, 18, 19, 27] among others. Concretely: rather than committing to an arbitrary secret directly, the committer can produce a hiding commitment to an ephemeral random secret vector (or secret polynomial), and then use the homomorphic property to construct a non-hiding proof of the correct opening of a linear combination of the desired secret and the ephemeral random secret. We omit the details here and refer interested readers to [7, §3.3] for more information.

Definition A.1. A deterministic *vector commitment scheme*

$$\mathcal{V} = (\text{vSetup}, \text{vCom}, \text{vGen}, \text{vVerify})$$

comprises four algorithms that operate as follows:

- $\text{vSetup}(1^\kappa, U, L) \to \bar{pp}$ is given a security parameter κ, a set U, and a maximum vector length L. It generates public parameters $\bar{pp}$.
- $\text{vCom}(\bar{pp}, \vec{v}) \to C$ is given a vector $\vec{v} \in U^\ell$ where $\ell \leq L$. It outputs a commitment string C.
- $\text{vGen}(\bar{pp}, \vec{v}, i) \to \pi_i$ is given a vector $\vec{v}$ and an index i. It outputs a proof string π_i.
- $\text{vVerify}(\bar{pp}, C, u_i, \pi) \to \text{True/False}$ takes as input a vector commitment C, an indexed element $u_i \in U$, and a proof

string π. It outputs True if $u_i = \vec{v}[i]$ and π is a witness to this fact and False otherwise.

Security guarantees for commitments. We begin by discussing the security requirements for polynomial commitments. Just like any kind of commitment scheme, a polynomial commitment must satisfy correctness, binding, and hiding. We make two special requirements for polynomial commitments: first, we want the commitment to bind to a particular polynomial and to its (max) degree, and second, the hiding property only needs to hold for a random polynomial for the reason stated above. The specific definitions below are taken from Alhaddad et al. [7] since they also focus on the setting of polynomial commitments that are deterministic and homomorphic.

Definition A.2 (Strong correctness). Let $\mathrm{pp} \leftarrow \mathrm{Setup}(1^\kappa, \mathbb{F}, D)$. For any polynomial $\phi(x) \in \mathbb{F}[x]$ of degree d with associated commitment $\hat{\phi} = \mathrm{pCom}(\mathrm{pp}, \phi, d)$:

- If $d \leq D$, then for any $i \in \mathbb{F}$ the output $y \leftarrow \mathrm{Eval}(\mathrm{pp}, \hat{\phi}, i)$ of evaluation is successfully verified by $\mathrm{Verify}(\mathrm{pp}, \hat{\phi}, y, d)$.
- If $d > D$, then no adversary can succeed with non-negligible probability at creating a commitment $\tilde{\phi}$ that is successfully verified at $d + 1$ randomly chosen indices.

Definition A.3 (Evaluation binding). Let $\mathrm{pp} \leftarrow \mathrm{Setup}(1^\kappa, \mathbb{F}, D)$. For any PPT adversary $\mathcal{A}(\mathrm{pp})$ that outputs a commitment $\tilde{\phi}$, a degree d, and two evaluations $y = \langle i, j, w \rangle$ and $y' = \langle i', j', w' \rangle$, there exists a negligible function $\varepsilon(\kappa)$ such that:

$$\Pr[(\tilde{\phi}, y, y', d) \leftarrow \mathcal{A}(\mathrm{pp}) : i = i' \wedge j \neq j' \wedge \mathrm{Verify}(\mathrm{pp}, \tilde{\phi}, \tilde{y}, d) \wedge \mathrm{Verify}(\mathrm{pp}, \tilde{\phi}, \tilde{y}', d)] < \varepsilon(\kappa).$$

Definition A.4 (Degree binding). Let $\mathrm{pp} \leftarrow \mathrm{Setup}(1^\kappa, \mathbb{F}, D)$. For any PPT adversary $\mathcal{A}$ that outputs a polynomial ϕ of degree $\deg(\phi)$, evaluation $\tilde{y}$, and integer d, there exists a negligible function $\varepsilon(\kappa)$ such that:

$$\Pr[(\phi, \tilde{y}, d) \leftarrow \mathcal{A}(\mathrm{pp}), \hat{\phi} = \mathrm{pCom}(\mathrm{pp}, \phi, \deg(\phi)) : \mathrm{Verify}(\mathrm{pp}, \hat{\phi}, \tilde{y}, d) \wedge \deg(\phi) > d] < \varepsilon(\kappa).$$

Definition A.5 (Hiding for random polynomials). Let $\mathrm{pp} \leftarrow \mathrm{Setup}(1^\kappa, \mathbb{F}, D)$, d be an arbitrary integer less than D, and $I \subset \mathbb{F}$ be an arbitrary set of indices with $|I| \leq d$. Randomly choose a $\phi \leftarrow \mathbb{F}[x]$ of degree d and construct its commitment $\hat{\phi} = \mathrm{pCom}(\mathrm{pp}, \phi, d)$. For all PPT adversaries $\mathcal{A}$, there exists a negligible polynomial $\varepsilon(\kappa)$ such that:

$$\Pr[(x, y) \leftarrow \mathcal{A}(\mathrm{pp}, \hat{\phi}, \{\mathrm{Eval}(\mathrm{pp}, \phi, i)\}_{i \in I}) : y = \phi(x) \wedge x \notin I] < \varepsilon(\kappa),$$

where the probability is taken over $\mathcal{A}$'s coins and the random choice of ϕ.

Finally, we discuss the security requirements for vector commitments. We omit a formal specification here because they are analogous to Definitions A.2, A.3, and A.5 above, but with polynomial commitments and proofs replaced with vector commitments and proofs. That is: correctness requires that honestly-created opening proofs will verify, evaluation binding requires that the adversary cannot find two openings to the vector commitment at the same index i that will both verify, and hiding requires that there is a negligible probability that an adversary can produce an opening proof corresponding to a vector commitment for a random vector that the adversary was never given. There is no equivalent to degree binding for vector commitments; only evaluation binding is required.

B SECURITY ANALYSIS

In this section, we prove that our Haven++ construction achieves the properties of a dual-threshold ACSS.

Proof of Liveness. If the dealer P_d is honest, then P_d will send everyone the same univariate polynomial commitments. Also, each party i receives one evaluation at index i on every univariate polynomial (a row polynomial). All of the checks on line 10 pass, so the honest parties can vector commit to the same list of polynomial commitments and produce the same vector commitment C along with the same proofs of inclusion for the polynomial commitments. This will enable every party to echo the right evaluations and corresponding proofs to everyone. As a result, all honest parties will pass the checks on lines 17 and interpolate their own univariate column polynomial and will be able to send ready messages with the right evaluations and proofs. This will enable every honrest party to reconstruct their row polynomial from the correct ready messages linked to the vector commitment C send by at least $n - t$ honest parties. If any dishonest party tries to send a malformed commitment or evaluation in their echo message or ready, then the evaluation binding property of the underlying commitment (cf. Definition A.3) ensures that it will not link back to the same root commitment, so honest parties will eventually disregard this message. Finally, if an honest party completes dispersal and invokes Reconstruct(j), then every honest party $(n - t)$ will evaluate their row polynomial at j and return it. This enables the honest party to run the online error correcting and reconstruct the univariate polynomial of degree $2t$ since there is only t possible errors that can happen.

Proof of Secrecy. Without loss of generality let's assume dispersal has been be done with t packed secrets. We will first analyze what the attacker learns during dispersal and then look at the reconstruction step with an inductive approach.

During dispersal, each party's view consists of n polynomial commitments, one row and one column polynomial together with evaluation proofs. As a result an attacker that can corrupt t parties has access to t full rows and t full columns. The hiding property of the polynomial commitments (Def. A.5) guarantees that this is insufficient to distinguish any other point on the column polynomial from random with non-negligible probability. It is left to show that the evaluations themselves don't leak information.

Let us first consider the information available for the attacker after dispersal has finished. The attacker has t points on every column polynomial (including the ones holding the secret) and knows t full columns. This is because the dealer sends a row polynomial to every party and during echo every party helps every other party reconstruct its column. Although the attacker knows t points on each column polynomial of degree t. Column-wise, Shamir secret sharing guarantees that the Adv learns nothing about any particular secret, unless they will learn one more point on that column. As a result column-wise from that information alone, a packed secret i is safe. Row-wise the attacker has t points on every row polynomial including the row 0 polynomial of degree $2t$ holding the $t+1$ packed secrets. For row 0, Shamir secret sharing also guarantees that the Adv learns nothing about any of the $t+1$ packed secrets. This is because information theoretically, it is easy to see that even if the attacker holds the t shares of a degree $2t$ polynomial, the other $t+1$ points are still indistinguishable from random. Ergo, the packed secrets could be anything. For example, let the packed secrets be the vector of all 0, interpolate a new polynomial made of degree $2t$ where $t+1$ points have the value 0 (the packed points), and the other t points are the attacker's points.

Let us consider the new information learned by the attacker when the reconstruction algorithm for the first packed secret i is called. The attacker learns the column polynomial i of degree t. i.e the attacker learns a new point on every row polynomial of degree $2t$, including the row 0 polynomial of degree $2t$ holding the $t+1$ packed secrets. Still, information theoretically the other t packed secrets are still indistinguishable from random. Even if the attacker holds $t+1$ shares of a degree $2t$ polynomial the other t points are free. Ergo, the other t packed secrets could still be anything.

Let us consider the new information learned by the attacker when the reconstruction algorithm is called the t^{th} time. The attacker has learned t column polynomials each of degree t on top of the ones they learned from dispersal. i.e the attacker has learned $2t$ points on every row polynomial of degree $2t$, including the the row 0 polynomial of degree $2t$ holding the 1 secret left. Still, information theoretically the last packed secrets is still indistinguishable from random. Even if the attacker holds $2t$ shares of a degree $2t$ polynomial, there is one point that is free. Ergo, the last packed secret could still be anything.

Lemma B.1. *Let $\phi_1 \dots \phi_n$ be a list of column polynomials of degree t. Suppose there exists a set $S \subset \{1, \dots, n\}$ of size $t+1$ such that for all $i \in S$, the row polynomial formed by interpolating $\phi_1(i), \phi_2(i), \dots, \phi_n(i)$ is of degree p. Then, there exists a unique bivariate polynomial $f(x,y)$ of degree p in one dimension and t in the other dimension where $\phi_i(.) = f(., i)$.*

Proof. Let $S = \{x_1, \dots, x_{t+1}\}$. Let $\psi_{x_1} \dots \psi_{x_{t+1}}$ be the row polynomials of degree p given by the statement of the lemma. Define $f(x,y) = \sum_{i=1}^{t+1} \psi_{x_i}(y) L_i(x)$, where $L_i(x)$ the appropriate Lagrange coefficient (namely, the unique degree-t univariate polynomial that vanishes at x_j for $i \neq j$ and is 1 at x_i).

Observe that f is of degree p in one variable and t in the other. Now we need to prove $f(x,y) = \phi_y(x)$. If $x \in S$, then this is true by construction, because $f(x,y) = \psi_x(y)$ (because there is only Lagrange coefficient that doesn't vanish at x), which is equal to $\phi_y(x)$ by definition of ψ. Since $f(\cdot, y)$ and $\phi_y(\cdot)$ are degree-t polynomials that agree on $t+1$ points (namely, all points in S), they must be equal as polynomials, and thus the statement is true for all x, not just $x \in S$.

We now need to prove uniqueness. Observe that for every i, $f(x,i)$, as a polynomial of degree less than n in x, is unique if it agrees with $\phi_i(x)$ in n points (because two different univariate polynomials of degree less than n cannot agree on n points). Thus, viewing $f(x,y)$ as a univariate polynomial in y that evaluates to polynomials in x, we know that its n evaluations are unique. Since it it has degree less than n in y, it must also be unique.

□

Proof of Correctness. Correctness states that all nonfaulty parties who complete reconstruction of the k^{th} secret should agree on the same secret, which in turn should be the same as the one used by the dealer if it was honest.

We reason about correctness in the following steps. First, our use of Bracha's broadcast ensures that all honest parties have agreement over the root commitment by the end of the sharing phase. Second, for the broadcast to succeed, at least $t+1$ honest parties must have received the actual vector of polynomial commitments in the dealer's sharing phase (or else they would not have echoed the root commitment, or anything else for that matter) and have checked that it forms a bivariate polynomial (lemma B.1). These parties collectively hold enough data to reconstruct the secrets. Moreover, in Algorithm 1 they will provide every honest party with their column polynomial and its commitment. Finally, this implies that the honest parties have enough information at reconstruction for the online error correction to terminate and produce the correct secret.

C PRIVATE POLYNOMIAL COMMITMENTS

Private polynomial commitments (as defined in §2.4) can be instantiated using zk-SNARKs. However, we make the observation that Bulletproofs [18] already supports this primitive, with a small modification. In fact, we can construct private polynomial commitments from any additively homomorphic polynomial commitment such that the same field is used to specify both the polynomial and the exponents of a Diffie-Hellman group. We describe the main ideas below, and show the full construction in Algorithm 5.

Bulletproofs demonstrate the application of inner product arguments to construct polynomial commitments. To verify a polynomial f's evaluation at a point i, the prover discloses an inner product involving two vectors, v_1 and v_2. Here, v_1 represents the coefficients of the polynomial f. The evaluation point i is exponentiated across a range from 0 to d, with d being the polynomial's degree, to form v_2: $v_2 = \langle 1, i, i^2, \dots, i^d \rangle$. Employing this technique reveals details about the coefficients within the vector v_1. To make it confidential, a standard technique [18] would be to ask the verifier for a challenge c, the prover sends both $f(i)$ and $f'(i)$ and proves the evaluation for $(f+cf')(i)$ instead of $f(i)$ where f' is picked uniformly at random from the field. Hence, v_1 contains information about the coefficients of $f+cf'$ instead of f. The same methods can be used to build private polynomial commitments on top of discrete log systems. Instead of the prover sending $f(i)$ and $f'(i)$, the prover sends $g^{f(i)}$ and $g^{f'(i)}$ (standard Feldman Commitments) and proves the evaluation for $(f+cf')(i)$ as before. The verifier can check the proof for $(f+cf')(i)$ using the inner product argument and then check in the exponent that indeed $g^{f(i)} * (g^{f'(i)^c}) == g^{(f+cf')(i)}$. In this method, in the same way as the standard technique, it is acceptable to reveal information about $(f+cf')(i)$ because the polynomial f' serves as a one-time pad that hides f from the verifier. Soundness follows from the fully binding property of the Feldman commitments ($g^{f(i)}$, $g^{f'(i)}$ and $g^{(f+cf')(i)}$) and the correctness of the Polynomial Commitment (in this case Bulletproofs).

We show in Algorithm 5 a formal construction for the case of Bulletproofs, where the main modification occurs in lines 7 and 10. We believe the technique generalizes to work for any additively homomorphic polynomial commitment.

D ASYNCHRONOUS MPC

Just like Beaver triples [10], Dual Secret Sharing [11] is a pre-processing building block for performing multiplication gates in MPC. Our construction from Algorithm 6 instantiated with $b = 1$, can be used to generate distributed Dual Secret Shared randomness with no trusted dealer. To open the secret under a $2t$ threshold, one can use Algorithm 2, while to open the same secret under a t threshold, one can use Algorithm 3. We remind the reader how to use $\{t, 2t\}$ dual sharing to do multiplication in the honest but curious case. Note that support for multiplication with malicious security can be achieved, if additively homomorphic polynomial commitments are used.

Multiplication in the Honest But Curious Setting. Assume there are two secrets s_1 and s_2 secret shared using Shamir secret sharing, such that any t+1 parties can reconstruct either secrets. If each party i multiplies the two shares it has, this will result in a polynomial S of degree $2t$ where $S(0) = s_1 s_2$ and each party i has the share $S(i)$. If the parties need to do more multiplications, the parties need to lower the threshold from $2t$ to t while still maintaining the same constant coefficient. The trick involves a pre-processing step, given a random number r, secret shared among n parties using two random polynomials HR and R of degree $2t$ and t respectively, such that $HR(0) = R(0) = r$ and where each party i has the pair of shares $HR(i), R(i)$. Each party opens $HR(i) - S(i)$. It is okay to open this value because $HR(i)$ is random and will act as a blinding factor to the value $S(i)$. Given $2t$ shares, the polynomial $HR - S$ is opened at 0 in the clear. Each party i locally adds $R(i)$ to the constant $(HR - S)(0)$. It is clear to see that this will result with every party having shares of $s_1 s_2$ under a degree t polynomial.

Algorithm 5 Private Polynomial Commitments from Additively Homomorphic Polynomial Commitments

Require: ϕ is a polynomial of degree d where every coefficient is uniformly sampled from $\mathbb{Z}_p$, $i \in \mathbb{Z}_p$ the index that the polynomial need to be opened at, $g \in G$ of order p, and pp denotes the public parameters used by the polynomial commitment.

1: **P's input:** $(\mathbf{g}, \phi, i)$
2: **V's input:** $(\mathbf{g}, \hat{\phi}, i)$
3: $P : \phi' \leftarrow$ **random** $\in \mathbb{Z}_p^d$
4: $P \rightarrow V : \hat{\phi}' = \text{pCom}(\text{pp}, \phi', d)$
5: $V : c \leftarrow$ **random** $\in \mathbb{Z}_p$
6: $V \rightarrow P : c$
7: $P \rightarrow V : g^{\phi'(i)}, g^{\phi(i)}$ and $\langle i, (\phi + c\phi')(i), w_i \rangle = \text{Eval}(\text{pp}, (\phi + c\phi'), i)$
8: V computes:
9: $\widehat{\phi + c\phi'} = \text{Hom}(\text{pp}, \hat{\phi}, \hat{\phi}', c)$
10: **if** Verify(pp, $\widehat{\phi + c\phi'}$, $\langle i, (\phi + c\phi')(i), w_i \rangle$) and $g^{(\phi + c\phi')(i)} = (g^{\phi(i)}) * (g^{\phi'(i)^c})$ **then**
11:     **return** $V_{accepts}$
12: **else**
13:     **return** $V_{rejects}$

E ADDITIONAL FIGURES

Algorithm 6 below provides a complete specification of our algorithm to generate distributed randomness.

In Figure 7, we show a pictorial representation of the batching that occurs within our ACSS protocol.

Algorithm 6 Generating Distributed Randomness for party i

SHARING PHASE:
$S \leftarrow \{\}$
Sample b random secrets $s_1, \ldots, s_b \leftarrow \mathbb{Z}_q$
$\phi_1, \ldots, \phi_b$ = Haven++ $(s_1, \ldots, s_b)$ ▷ Let $\phi_0, \ldots, \phi_b$ be the column univariate polynomial used in Haven++ dispersal
$S \leftarrow S \cup \{j\}$ when j-th Haven++ dispersal terminates at party p_i
AGREEMENT PHASE:
if $|S| = 2t + 1$ **then**
    Let $S_i \leftarrow S$, invoke $MVBA(S_i)$ with predicate $P(S_j, S)$ ▷ S_j is the input value of some party p_j, S is party p_i's local variable defined in the Sharing Phase. $P(S_j, S)$ only returns 1 once $S_j \subseteq S$.
RANDOMNESS EXTRACTION PHASE:
Let T be the output of the $MVBA$ protocol after picking exactly the first $2t + 1$
Let $B_{row} = []$ ▷ used to store row i of every bivariate polynomial, there is going to be b of them
Let $B_{col} = []$ ▷ used to store column i of every bivariate polynomial, there is going to be b of them
Let $B_{com} = []$ ▷ used to store the column polynomial commitments of the new bivariate polynomials, there is going to be b of them. It is enough to store $2t + 1$ column polynomial commitments
Let $O_{col,1}, O_{col,n}$ be the columns that p_i has dispersed during the Haven++ dispersal and let $O_{col,-b}, O_{col,0}$ be the columns holding the packed secrets.
for each $j \in T$ **do**
    Let $O_{row,i}$ be row i that p_j has dispersed during the Haven++ dispersal.
    Let $\hat{\phi}_1 \ldots \hat{\phi}_n$, be all n column polynomial commitments that has been dispersed by p_j during the Haven++ dispersal.
    Compute $\hat{\phi}_{-b} \ldots \hat{\phi}_0$ homomorphically from $\hat{\phi}_1 \ldots \hat{\phi}_{2t+1}$
    for each $k \in [-b, 0]$ **do**
        $B_{com}[-k].append(\hat{\phi}_{-k})$
        $B_{col}[-k].append(O_{col,-k})$
        $B_{row}[-k].append(O_{row,j}(k))$
        if $|B_{row}[-k]| = 2t + 1$ **then**
            $B_{row}[-k] = interpolate(B_{row}[-k])$
    for each $j \setminus T$ **do** ▷ only needed if the higher threshold of every bivariate need to be opened
        for each $k \in [-b, 0]$ **do**
            send $(-k, row_i(j))$ to P_j ▷ ensure that all parties have columns
DISTRIBUTION PHASE:
if $i \setminus T$ **then** ▷ only needed if the higher threshold of every bivariate need to be opened
    Upon receiving((k, val) from party P_m for the first time with k:
        $B_{col}[k].append(m, val)$
        if $|B_{col}[k]| \geq t + 1$ **then** ▷ Run Online Error correcting code
            $\tilde{\phi_{col}} = ECC(B_{col}[k], t, e_k)$ ▷ with e_k initialized to 1, attempt to interpolate the column polynomial of degree t
            if $\tilde{\phi_{col}} \neq \perp$ **then**
                $B_{col}[k] = \tilde{\phi_{col}}$
            $e_k = e_k + 1$ ▷ increase the number of errors by one with each failed decoding
        if for all $k \in [-b, 0]$ $B_{col}[k]$ is a polynomial of degree t **then**
            $output(B_{com}, B_{col}, B_{row})$
else
    $output(B_{com}, B_{col}, B_{row})$

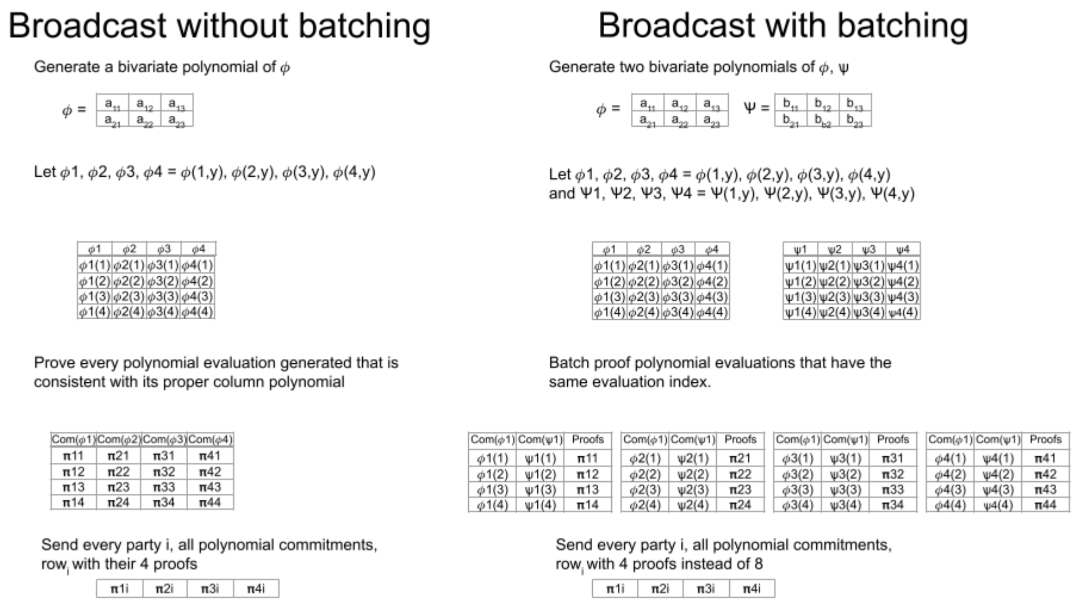

Figure 7: The broadcast phase of Haven++ with and without batching when the batch size is 2 and $t = 1$. Regardless of the batch size, the dealer always produces n proofs for every party.