Asynchronous Distributed Key Generation for Computationally- Secure Randomness, Consensus, and Threshold Signatures.

ELEFTHERIOS KOKORIS-KOGIAS*, Faceook Novi & IST Austria
DAHLIA MALKHI, Faceook Novi
ALEXANDER SPIEGELMAN, Faceook Novi

In this paper, we present the first *Asynchronous Distributed Key Generation* (ADKG) algorithm which is also the first distributed key generation algorithm that can generate cryptographic keys with a dual $(f, 2f+1)$−threshold (where f is the number of faulty parties). As a result, using our ADKG we remove the trusted setup assumption that the most scalable consensus algorithms make. In order to create a DKG with a dual $(f, 2f+1)$− threshold we first answer in the affirmative the open question posed by Cachin et al. [7] on how to create an Asynchronous Verifiable Secret Sharing (AVSS) protocol with a reconstruction threshold of $f+1 < k \leq 2f+1$, which is of independent interest. Our *High-threshold-AVSS* (HAVSS) uses an asymmetric bivariate polynomial to encode the secret. This enables the reconstruction of the secret only if a set of k nodes contribute while allowing an honest node that did not participate in the sharing phase to recover his share with the help of $f+1$ honest parties.

Once we have HAVSS we can use it to bootstrap scalable partially synchronous consensus protocols, but the question on how to get a DKG in asynchrony remains as we need a way to produce common randomness. The solution comes from a novel *Eventually Perfect Common Coin* (EPCC) abstraction that enables the generation of a common coin from n concurrent HAVSS invocations. EPCC's key property is that it is eventually reliable, as it might fail to agree at most f times (even if invoked a polynomial number of times). Using *EPCC* we implement an *Eventually Efficient Asynchronous Binary Agreement* (EEABA) which is optimal when the EPCC agrees and protects safety when EPCC fails.

Finally, using EEABA we construct the first ADKG which has the same overhead and expected runtime as the best partially-synchronous DKG ($O(n^4)$ words, $O(f)$ rounds). As a corollary of our ADKG, we can also create the first Validated Asynchronous Byzantine Agreement (VABA) that does not need a trusted dealer to setup threshold signatures of degree $n-f$. Our VABA has an overhead of expected $O(n^2)$ words and $O(1)$ time per instance, after an initial $O(n^4)$ words and $O(f)$ time bootstrap via ADKG.

1 INTRODUCTION

A common assumption made by many modern Byzantine fault tolerant distributed algorithms is the existence of a trusted dealer that generates and distributes cryptographic keys at the beginning of every execution. For example, efficient asynchronous Byzantine agreement protocols [1, 3, 9, 17, 29] use a shared coin scheme to produce randomness [34], efficient state machine replication protocols [20, 35] use a threshold signature scheme to reduce communication complexity, and efficient secure multiparty computation protocols [22, 23] use threshold encryption [26] to reduce the communication complexity for sharing secret inputs. All these schemes require a trusted dealer, which is a single point of failure and a potential weakness for secure decentralized systems.

It is therefore natural to ask under what network assumptions and at what cost the requirement of a trusted dealer can be substituted with a distributed key generation (DKG) protocol. A DKG protocol allows a group of parties to distribute private shares of a cryptographic key and later use them to compute a common value such that an adversary controlling a threshold of the parties cannot predict the value. Thereby, this value can be used to produce unpredictable randomness or as a "private" key.

*Corresponding Author.

Authors' addresses: Eleftherios Kokoris-KogiasFaceook Novi & IST Austria, lefteris2k@gmail.com; Dahlia MalkhiFaceook Novi, dahliamalkhi@gmail.com; Alexander SpiegelmanFaceook Novi, sasha.spiegelman@gmail.com.

In synchronous communication settings, a DKG protocol can be realized via a combination of two building blocks, secret sharing and consensus [32] (or a broadcast channel such as a blockchain [2, 18]). In a nutshell, all parties simultaneously choose and share a secret and then use a Byzantine agreement instance for each secret in order to agree if it should be part of the key. The key is the sum of all valid secrets and the share of each party is the sum of the corresponding shares. To the best of our knowledge, no asynchronous DKG (ADKG) protocol has been previously proposed. We focus on protocols with $n = 3f+1$ parties that assume no trusted setup except for public key infrastructure (PKI). We further explore protocols that support threshold recovery of $2f + 1$, which is required by efficient Byzantine agreement algorithms that use threshold signatures to reduce the size of the messages from linear in the number of parties to constant [1, 20, 35].

A naive approach for ADKG is to apply the ideas in [32] to the asynchronous settings. For example, it is possible to use the AVSS scheme of Cachin et al [7] and n independent parallel instances of a binary agreement protocol[1] like [4, 6]. However, the resulting algorithm has three drawbacks: First, the secret sharing in [7] has a reconstruction threshold of $f + 1$ and thus the resulting ADKG cannot have the desired $2f + 1$ threshold. Second, running n binary agreements does not guarantee a successful protocol execution, since they can all terminate with 0 which means that the key will include no secrets. Finally, even if we could guarantee that more than f instances terminate successfully[2], the resulting protocol would be inefficient with a communication complexity of $O(n^5 logn)$.

In this paper we present the first ADKG protocol with a recovery threshold of $2f + 1$ and low communication cost. Formally, the main theorem we prove in this paper is following:

Theorem 1.1. *There exists a protocol among n parties that solves Asynchronous Distributed Key Generation (ADKG) with reconstruction threshold $k \leq n - f$ and is secure against an adaptive adversary that controls up to $f < n/3$ parties, with expected $O(n^4)$ communication complexity and expected $O(f)$ running time.*

In a nutshell, our protocol follows the idea of concurrently sharing n secrets and then agree which to consider for the key. However, instead of using a costly Byzantine agreement instance for each secret, we use the secrets as the driving randomness source to build an efficient common coin which in turn we use for an efficient Byzantine agreement. In particular, we observe that to build a common coin from the secrets we can use a slightly weaker agreement notion which is not subject to the FLP impossibility result. To this end, we first improve the asynchronous secret sharing scheme in [7] to support $2f + 1$ reconstruction threshold. Then, we rely on the completeness property of our secret sharing scheme and guarantee that eventually all honest parties get shares for the same secrets. As a result, we know that all parties eventually agree on the set of secrets and ,hence, could use it for a shared coin. Unfortunately, the parties do not know when this happens (in contrast to the agreement problem) and cannot ever terminate.

To circumvent this non-termination problem, our idea is to let the parties optimistically think that every received share is the last one (i.e., all correct AVSS instance have terminated and all other instances are faulty) and try to terminate. Each time a new share is received by $n - f$ parties, they generate new key shares and initiate a shared coin protocol (produce a threshold signature and hash it to get unpredictable randomness). This shared coin flip is in turn used in some efficient binary agreement protocol (these keys replace the ones produced by the trusted dealer). If the parties happen to agree on the key (their sets of shares correspond to the same secrets), then the Byzantine agreement protocol terminates successfully. Otherwise, some parties received shares that others have not yet received and they will try again to terminate when the next (additional) secret is recoverable by all honest parties. We call our

[1] Each instance agrees on whether an AVSS secret is correctly shared.
[2] So that the adversary does not know all the secrets that are included in the key.

coin *Eventually Perfect Common Coin (EPCC)* and the resulting Byzantine agreement *Eventually Efficient Asynchronous Binary Agreement (EE-ABA)* , because eventually (after at most f failed tries) the protocols converge to the optimal solutions.

Finally, once we have an EE-ABA, we run n instances that share the same EPCC and use it in order to decide on the final set of shares, which terminates the ADKG protocol. In order to guarantee that the final key is unpredictable, the parties refrain from voting 0 in the binary agreement instances that they consider faulty until they witness $f + 1$ binary agreements terminating with 1 (which is guaranteed to happen due to the strong termination of the HAVSS). Next we explain the algorithms in more detail and prove that parties cannot disagree on the set of shares more than f times.

1.1 Technical contribution

We break the ADKG construction in a bottom-up manner, starting with a building block (Section 3) we call *High-threshold Asynchronous Verifiable Secret Sharing* (HAVSS). HAVSS is an extension of Cachin et al. [7] AVSS protocol that answers in the affirmative the open question they posed on the existence of an AVSS protocol that has a reconstruction threshold of $f + 1 < k \leq 2f + 1$. To achieve this, we separate the reconstruction threshold (which we increase to k) from the recovery threshold (which is still $f + 1$). In order to encode this change, we use an *asymmetric* bivariate polynomial where each dimension plays a different role (recovery, reconstruction) and we defend against an adaptive adversary with a reliable broadcast step before terminating the sharing. More formally HAVSS satisfies the following lemma.

Lemma 1.2. *There exists a protocol among n parties that solves Asynchronous Verifiable Secret Sharing (AVSS) for reconstruction threshold $f + 1 < k \leq n - f$, with no trusted setup, and is secure against an adaptive adversary that controls up to $f < n/3$ parties, with $O(n^3)$ word communication.*

The "secret sauce": The second (intermediate) building block is the *weak Distributed Key Generation* (Section 4). It builds on top of n parallel HAVSS invocations and uses the fact that all honest nodes eventually terminate all correct HAVSS to deliver a prediction on what the DKG should output. The wDKG is weaker than consensus because it refrains from outputting a final decision. Instead, it acts as an eventually perfect agreement detector. Any protocol that uses the wDKG gets the guarantee that eventually all parties will output the same key, but the specific time when the detector becomes perfect cannot be determined. One key property of wDKG is that every prediction is a superset of all prior predictions, hence there can only be a limited, totally-ordered number of predictions.

Our third building block (Section 5) is called *Eventually Perfect Common Coin* (EPCC). It relies on the wDKG to detect the points of agreement and on adaptively secure deterministic threshold signatures [28] to produce the randomness. The key property of the EPCC is that the adversary can only force it to disagree a finite (f) number of times. This happens because a point of disagreement occurs only if $f + 1$ honest parties are slower than the rest and the adversary brings them up to speed after they have invoked the EPCC but before they deliver the result. Due to the way the wDKG is constructed this can happen for at most f different keys and for each candidate key it may happen at most once.

Once we have the EPCC we can use the protocol of Moustefaoui et al. [29] to create our fourth building block: an efficient Asynchronous Binary Agreement protocol that does not assume a trusted setup (Section 6.1). We call it *Eventually Efficient ABA (EEABA)* as it might have f failed runs before converging, but once it converges it is optimal (with communication complexity of $O(n^2)$ and constant expected round complexity). Formally EEABA achieves the following:

Lemma 1.3. *There exists a protocol among n parties that solves Asynchronous Binary Agreement (ABA) without a trusted dealer in the authenticated setting and is secure against an adaptive adversary that controls up to $f < n/3$ parties, with $O(n^4)$ one-shot ($O(n^2)$ amortized) word communication and expected $O(f)$ one-shot ($O(1)$ amortized) running time.*

Finally, in Section 6.2 we invoke n concurrent and correlated EEABAs (one for every HAVSS) to agree on the set of shares that construct the key and complete the ADKG protocol (Theorem 1).

Corollaries. Since solving DKG implies a solution for consensus (if the secret value is public then it can be used as the consensus decision), a corollary of our main theorem is:

Corollary 1.4. *There exists a protocol among n parties that solves Validated Asynchronous Byzantine Agreement without a trusted dealer in the authenticated setting and is secure against an adaptive adversary that controls up to $f < n/3$ parties, with expected $O(n^4)$ word communication and expected $O(f)$ running time.*

And through the combination of our ADKG with the optimal validated asynchronous Byzantine agreement (VABA) of [1] a second corollary is:

Corollary 1.5. *There exists a protocol among n parties that solves Validated Asynchronous Byzantine Agreement without a trusted dealer in the authenticated setting and is secure against an adaptive adversary that controls up to $f < n/3$ parties, with expected $O(n^2)$ amortized word communication and expected constant amortized running time.*

Contributions. In summary our contributions are:

- We answer the open problem of a high-threshold AVSS posed by Cachin et al. [7] affirmatively. HAVSS in combination with Hybrid-DKG [24] removes the setup requirement of efficient partially synchronous consensus protocols [20, 35].
- We introduce a novel EPCC construction that disagrees at most f times but can be used polynomially many times.
- Using our EPCC inside the protocol of Moustefauoi et al. [29] we create EEABA protocol that needs no trusted setup. EEABA is optimal if amortized. It terminates in $O(f)$ one-shot ($O(1)$ amortized) expected rounds and has $O(n^4)$ for one-shot, ($O(n^2)$ amortized) word complexity.
- Using n parallel invocation of Binary Agreement (all sharing the same EPCC), we construct a computationally-secure, efficient, leaderless ADKG. Once the ADKG terminates, we can use the resulting key as a perfect common coin and as the key used in the threshold signature scheme, which are the building blocks of VABA. The ADKG has $O(n^4)$ word complexity and terminates in an expected $O(f)$ rounds. Hence, the combination of ADKG and VABA results in the first trustless VABA solution, which is also optimal if amortized.

1.2 Related work

Consensus is one of the most well studied distributed systems problem, first introduced by Pease et al [31], which has become once again relevant due to the interest in blockchain protocols [25, 27]. The problem can be stated informally as: how to ensure that a set of distributed processes achieve agreement on a value despite a fraction of the processes being faulty. From a theoretical point of view, the relevance of the consensus problem derives from several other distributed systems problems being reducible or equivalent to it. Examples are atomic broadcast [21], or state machine replication [33]. Algorithms that solve consensus vary much depending on the system model. This paper considers a

message-passing setting for systems that may experience Byzantine (or arbitrary) faults in asynchronous settings (i.e., without timing assumptions).

In this paper, we focus on 3 interconnected variants: Asynchronous Binary Agreement (ABA), Distributed Key Generation (DKG), and Validated Asynchronous Byzantine Agreement (VABA). Furthremore, we survey Asynchronous Secure Multiparty Computation (AMPC) that could provide a generic solution to our problem.

ABA:. The first optimally resilient ($f < n/3$) ABA was introduced by Bracha [6]. It is based on locally drawn random coins used to defend against a network controlling adversary. As the protocol uses local randomization it can only terminate when all correct processes happen to propose the same (0 or 1) value which has an expected $O(2^n)$ number of rounds with every round costing $O(n^3)$ messages. Canetti and Rabin [11] where the first to propose an ABA that has polynomial total communication complexity, however, the protocol is far from practically efficient with a cost of $O(n^8 log n)$ bits. Advancements in the information-theoretic secure model have lowered the cost down to $O(n^6)$ [4].

In order to reduce the communication complexity, Cachin et al [9] demonstrated how to achieve consensus against a computationally-bounded adversary using cryptography. Trying to achieve this, however, introduced a new assumption of a trusted dealer that deals a perfect common-coin. Mostefaoui et al. [29] slightly weakened the assumption of Cachin et al. by assuming a weak common-coin. Nevertheless, it remains an open problem on how to get such a coin efficiently. This is the core of our work, we build an eventually perfect common coin without the need of a trusted dealer. Our coin is also in the computationally-bounded adversary and falls in-between the weak coin and the perfect coin and as a result, can power Mostefaoui's protocol.

DKG:. A distributed key generation is a protocol that is executed once by a set of parties in order to achieve consensus on a shared secret key. The core idea is that each party uses secret sharing to disperse some secret value and then the parties reach consensus on which secret values have been correctly shared. In the end, these values are combined and the final result is a threshold private-public key-pair that can be used for efficient ABA [9] and VABA [1]. The first DKG was proposed by Pedersen [32] and is fully synchronous. Gennaro et al. [19] showed that Pedersen's scheme is secure if used for threshold signatures, but does not produce uniformly random keys. Hence they also proposed a scheme that produces such keys, which is not of interest to our protocols. Later, Kate et al. [24] realized that synchronous protocols are not suitable for large scale deployment over the Internet and proposed a partially-synchronous DKG instead. Their protocol has a worst-case $O(n^4)$ bit complexity and produces keys with a threshold of $k = f + 1$.

Our contribution to the DKG space is two-fold. First, we show how to generate keys with threshold reconstruction $k = 2f + 1$, which as we already mentioned can be used to power scalable partially synchronous BFT protocols [20, 35]. Second, we create the first asynchronous DKG with $O(n^4)$ word complexity making it practical to generate distributed keys with no timing assumptions.

VABA:. The VABA problem was introduced by Cachin et al. [8] which generalizes ABA, by allowing any externally valid value to be eligible for consensus. In this model, Abraham et al. [1] have provided an optimal solution ($f < n/3$) for VABA that has an expected complexity of $O(n^2)$ messages and terminates with probability 1 in an expected constant number of rounds. Both these protocols assume a perfect-coin, hence require a trusted setup. Our contribution in this model is also two-fold. First we show how we can implement a VABA protocol with no trusted setup and second we show how to bootstrap the more efficient protocols [1, 8] with our ADKG in order to get an optimal VABA if we amortize the cost of the ADKG over $O(n^2)$ runs.

Secure Multiparty Computation: On a first glance our protocol can be categorised as a special case of Asynchronous Secure Multiparty Computation [5], however with further inspection it actually provides a foundation for increasing the efficiency [5, 14, 15, 22, 23, 30] and removing the trusted setup assumption [14, 22, 23] of existing multiparty computation protocols.

More specifically, existing MPC protocols assume access to a Byzantine Agreement black box which they need to reach agreement on the inputs by deploying n parallel BAs. However this black box deployment of BA leads to inefficiencies leading to an expected $O(n^5 logn)$ world complexity in the cryptographically secure setting. Using our protocol which opens the black boxes and reuses the common coin, we can agree on the same inputs in only $O(n^4)$.

Furthermore, MPC protocols either assume a trusted setup of threshold signatures and threshold encryption [14, 22, 23] or employ a special type of AVSS called ACSS [5], which guarantees that all honest parties (instead of $f + 1$) get a share. Our HAVSS provides the same guarantees, making it a cryptographically secure ACSS protocol. Choudry and Patra [13] have created a framework where an MPC protocol can be constructed using BA and ACSS, as a result if we plug in our HAVSS and couple it with error-corrected reliable broadcast [10] we could get the most efficient AMPC with complexity of $O(n^3 logn)$ per multiplication gate. This improvement comes at the cost of sacrificing unconditional security since the state of the art has an $O(n^5 logn)$ cost[3]. However, the most natural use of ADKG would be to bootstrap the threshold encryption and threshold signing protocols of [23] and then run at $O(n^2)$ cost per multiplication gate. If the AMPC protocol has more than $O(n^2)$ gates, then we can get an amortized cost of $O(n^2)$ per gate.

In summary, this paper provides practical improvements on the foundation protocols of AMPC which could result through composition to practically efficient protocols. However, we leave the actual secure implementation and proofs to future work.

2 MODEL AND DEFINITIONS

In order to reason about distributed algorithms in cryptographic settings we adopt the model defined in [9]. For space limitation and better readability we define here a simplified version and the full formal model can be found in [1, 8, 9] and in Appendix A. We consider an asynchronous message passing system consisting of a set Π of n parties and an adaptive adversary. The adversary may control up to $f < n/3$ parties during an execution. An adaptive adversary is not restricted to choose which parties to corrupt at the beginning of an execution, but is free to corrupt (up to f) parties on the fly. Note that once a party is corrupted, it remains corrupted, and we call it faulty. A party that is never corrupted is called *honest*.

Communication. We assume asynchronous authenticated links controlled by the adversary, that is, the adversary can see all messages and decide when and what messages to deliver but cannot deliver a message from an honest party that was not generated by it. In order to be able to use cryptographic tools in asynchronous settings, the model defined in [1, 8, 9] restricts the adversary to perform no more than a polynomial in the security parameter number of computation steps during the time a message between two honest parties is sent and delivered. For completeness, in Appendix A, we give the formal definition of the assumption on message delivery and the termination requirement in asynchronous protocols with computationally bounded adversaries. However, in order to be able to focus on the distributed computing aspect of our work, we assume throughout the paper perfect cryptographic tools, standard delivery assumptions and termination requirement. That is, we assume every message between two honest parties is eventually delivered.

[3]Concurrent non peer-reviewed works claims reduction to $O(n^4 logn)$ [12]

Complexity. Following [1], our basic communication unit is *word*, which may contain a constant number of values of some domain $\mathbb{V}$ and cryptographic signatures. We define the total *communication cost* of our protocol to be the number of words sent among honest parties. One word is a signature that is linear in the size of the security parameter.

Cryptographic Abstractions. Given that our protocols use cryptographic constructions as black boxes, we assume perfect cryptographic tools and present simplified educational examples that use the multiplicative notation and simple computationally hiding commitments. Furthermore, in order to still have a correct protocol we employ the Diffie-Hellman Based Threshold Coin-Tossing Scheme of Cachin et al. [9]. This way the reader can focus on the distributed aspect of the protocol which is the novelty. However, in order to be adaptively-secure, the actual implementation of our consensus algorithm requires pairing-based threshold cryptography, as shown by Libert et al. [28]. More specifically, Libert et al. runs a classic synchronous DKG [32], but we can instead use our ADKG (Section 6.2) to terminate their protocol in asynchrony and generate the consistent secret shares.

Diffie-Hellman Based Coin. In order to follow our protocols, we need to present the coin-tossing protocol of Cachin et al. [9]. We work with a group $\mathbb{G}$ of large prime order q. At a high level, the value of a coin C is obtained by first hashing C to obtain $\bar{g} \in \mathbb{G}$, then raising $\bar{g}$ to a secret exponent $x_0 \in \mathbb{Z}_q$ to obtain $\bar{g_0} \in \mathbb{G}$, and finally hashing $\bar{g_0}$ to obtain the value $F(C) \in \{0, 1\}$.

In this paper, we distributively generate the secret exponent x_0 such that before the coin-toss is invoked every party P_i holds a share x_i of x_0. The party uses this share to generate a share of the coin $F(C)$ which is $\bar{g}^{x_i}$. For our purpose we abstract the inner workings of the coin by exposing four functions:

generate-share(x_i, C), it uses the partial key x_i to generate a coin-share for coin C.

verify-share(C, m, σ) verifies that σ is a valid share of party P_m.

generate-coin$(C, [\sigma_i])$ generates a coin given a threshold of valid shares of C.

verify-coin$(C, \sigma_{\mathbb{P}})$, verifies that the given value $\sigma_{\mathbb{P}}$ correspond to valid coin for C.

3 HIGH-THRESHOLD ASYNCHRONOUS VERIFIABLE SECRET SHARING

Existing AVSS [7, 11] schemes provide a reconstruction threshold up to $n - 2f$ shares. Intuitively this is because at the sharing step the participating nodes can only wait for $n - f$ ready message from nodes, where ready confirms that a node has verified its share. As a result in the reconstruction phase, there can be up to f (corrupt) nodes who participated at the sharing but do not participate in the reconstruction, hence for the reconstruction to succeed the recovery threshold should be $n - f - f = n - 2f$.

In this section we present our HAVSS scheme that requires a high threshold of up to $n - f$ shares for the secret reconstruction. Our scheme is an extension of the AVSS scheme by Cachin et al. [7], where the dealer uses an *asymmetric* bivariate polynomial instead of a symmetric one. The key idea is that one dimension of the asymmetric bivariate polynomial has an order of f and is used for shares recovery, while the other dimension has an order of $2f$ and is used for the secret reconstruction.

3.1 Definition

Our protocol falls in the class of *dual-threshold sharing* [9], which are protocols that allow the reconstruction threshold of a secret to be more than $f + 1$. Although in the original AVSS [7] paper the authors introduce the notion of a

dual-threshold secret sharing scheme with reconstruction threshold up to $n - f$, the AVSS described only works for reconstruction threshold $n - 2f$. In this work, we solve the open problem posed by the authors on creating an (n, k, f) dual-threshold AVSS where $f + 1 < k \leq n - f$. This is an important challenge since an $(f, n - f)$-AVSS can power[4] efficient Byzantine agreement [1, 35] and efficient MPC [22, 23] which currently require a trusted dealer during setup.

We follow the definitions of Cachin et al [7] and modify them for HAVSS: A protocol with a tag $ID.d$ to share a secret $s \in \mathbb{Z}_q$ consists of a *sharing* stage and a *reconstruction* stage as follows.

Sharing stage. The sharing stage starts when the party initializes the protocol. In this case, we say the party *initializes a sharing ID.d*. There is a special party P_d, called a *dealer*, which is activated additionally on an input message of the form $(ID.d, \text{in}, \text{share}, s)$. If this occurs, we say P_d *shares* s *using* $ID.d$ among the group. A party is said to *complete the sharing* $ID.d$ when it generates an output of the form $(ID.d, \text{out}, \text{shared})$. An honest but slow party might not complete the sharing if the dealer is malicious. In this case, it can still recover its share of the secret from the rest of the parties that managed to terminate the sharing. Such a party is said to *indirectly complete the sharing* $ID.d$.

Reconstruction stage. After a party has completed the sharing, it may be activated on a message $(ID.d, \text{in}, \text{reconstruct})$. In this case, we say the party *starts the reconstruction for* $ID.d$. At the end of the reconstruction stage, every party should output the shared secret. A party P_i terminates the reconstruction stage by generating an output of the form $(ID.d, \text{out}, \text{reconstructed}, z_i)$. In this case, we say P_i *reconstructs* z_i *for* $ID.d$. This terminates the protocol.

Furthermore, the protocol should satisfy the following properties for our threat model, except with negligible probability:

H(i) : Liveness. If the adversary initializes all honest parties on sharing $ID.d$, delivers all associated messages, and the dealer P_d is honest throughout the sharing stage, then all honest parties complete the sharing. Moreover, if all honest parties subsequently start the reconstruction for $ID.d$, then every honest party P_i reconstructs some z_i for $ID.d$.

H(ii) : Agreement. Provided the adversary initializes all honest parties on sharing $ID.d$ and delivers all associated messages, the following holds: If some honest party completes the sharing $ID.d$, then all honest parties will complete the sharing of $ID.d$.

H(iii) : Correctness. Once k honest parties have completed the sharing of $ID.d$, there exists a fixed value z such that the following holds:

(1) If the dealer has shared $(ID.d, \text{in}, \text{share}, s)$ and is honest throughout the sharing stage then $z = s$.
(2) If an honest party P_i reconstruct z_i for $ID.d$ then $z_i = z$.

H(iv) : Privacy. If an honest dealer shared $(ID.d, \text{in}, \text{share}, s)$ and less than $k - f$ honest parties have started the reconstruction for $ID.d$, then the adversary has no advantage when trying to guess the value s.

3.2 Implementation

The key mechanism of HAVSS (see Figure 1) is the use of an asymmetric bi-variate polynomial $(k - 1, f)$. The first dimension is used to protect the secret, which is reconstructed if k shares are combined, whereas the second dimension is used to enable recovery of the shares of the secret from any group of $f + 1$ honest participants.

Let p and q be two large primes satisfying $q \mid (p - 1)$, and $q > n$. Let $\mathbb{G}$ denote a multiplicative subgroup of order q of $\mathbb{Z}_p$ and let g be a generators of $\mathbb{G}$.

[4] Coupled with a suitable DKG [24]

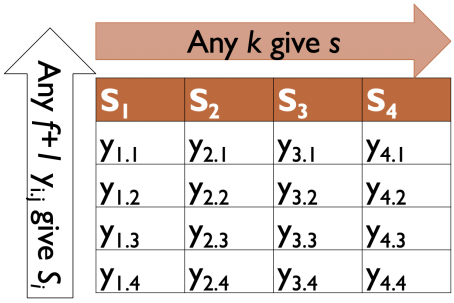

Fig. 1. Intuition of HAVSS. P_j receives row $y_{*.j}$ which is used to compute the recovery polynomial $b_j(y)$ and column $y_{j.*}$ which is used to compute the share polynomial $a_j(x)$ and recover its share $S_j = a_j(0)$. If a malicious dealer does not send P_m its share, P_m can still complete indirectly the sharing. This is possible because P_j, that completes the sharing directly, will send P_m a message with $y_{m.j}$. Since there are $f+1$ available parties that should have shares in column m and complete the sharing directly, P_m will get enough points to recover $a_j(x)$, hence recover $S_m = a_m(0)$. As a result, eventually k parties will have shares S_i, compute locally $u(0, x)$ and recover the secret $s = u(0, 0)$.

(1) The dealer computes a one-dimensional sharing of the secret and uses the second dimension of the bi-variate polynomial to share the secret-shares. This is achieved by choosing a random bivariate polynomial $u \in \mathbb{Z}_q[x, y]$ where the dimension $[x]$ is of degree $t = k - 1$ and the dimension $[y]$ is of degree f with $u(0, 0) = s$ and it commits to $u(x, y) = \sum_{j,l=0}^{t,f} u_{jl} x^j y^l$ by computing a commitment matrix $\mathbb{C} = \{C_{jl}\}$ with $C_{jl} = g^{u_{jl}}$ for $j \in [0, t]$, $l \in [0, f]$. The dealer sends each party P_i a message containing the commitment matrix $\mathbb{C}$ as well as a *recovery polynomial* $a_i(y) := u(i, y)$ of order f and a *share polynomial* $b_i(x) := u(x, i)$ of order t.
(2) When the parties receive the send message from the dealer, they echo the points in which their share and recovery polynomial overlap with each other. To this effect, P_i sends an echo message containing $\mathbb{C}$, $a_i(j)$, $b_i(j)$ to every party P_j.
(3) Upon receiving k echo messages that agree on $\mathbb{C}$ and contain valid points, every party P_i interpolates its own share and recovery polynomials $\bar{a}_i$ and $\bar{b}_i$ from the receiving points and verifies that they are the same as the ones received by the dealer. Then P_i sends a ready message containing $\mathbb{C}$.
(4) Once the party receives a total of $n - f$ ready messages that agree on $\mathbb{C}$, it *completes* the sharing. Its share of the secret is $s_i = \bar{a}_i(0)$. In order to guarantee that the rest of the parties also complete the sharing, it sends the set of $n - f$ ready messages (for the parties that send the ready message and will finish with shared) as well as $b_i(j)$ to every party P_j (for the ones that are slow and will finish indirectly).
(5) A party that has not sent a ready message yet, needs to consider the possibility that it is in the slow set. Hence, if it receives $f + 1$ consistent shared messages, it interpolates $s_i = \bar{a}_i(0)$ and finishes the sharing indirectly.

As a result, during reconstruction, every honest node eventually has a correct share of the secret. Hence eventually k points that are consistent with $\mathbb{C}$ become public. Once P_i receives them all, he can interpolate $u(0, y)$ and recover $s = u(0, 0)$. The protocol has communication complexity of $O(n^4)$, however, it can be optimized to $O(n^3)$ as shown in [7].

3.3 Protocols

Algorithm 1 and 2. In the protocol description, the following predicates are used:

verify-poly($\mathbb{C}, i, a, b$), where a, b are polynomials of degree f and t respectively, i.e.,

$$a(y) = \sum_{l=0}^{f} a_l y^l \quad \text{and} \quad b(x) = \sum_{j=0}^{t} b_j x^j$$

This predicate verifies that the given polynomials are share and recovery polynomials for P_i consistent with $\mathbb{C}$; it is true if and only if for $l \in [0, f]$, it holds $g^{a_l} = \prod_{j=0}^{f} (\mathbb{C}_{jl})^{i^j}$ and for $j \in [0, t]$, it holds $g^{b_l} = \prod_{l=0}^{t} (\mathbb{C}_{jl})^{i^l}$.

verify-point($\mathbb{C}, i, m, \alpha, \beta$), verifies that the given values α, β correspond to points $f(m, i)$, $f(i, m)$, respectively, committed to $\mathbb{C}$, which P_i supposedly receives from P_m; it is true if and only if $g^{\alpha} = \prod_{j,l=0}^{f,t} (\mathbb{C}_{jl})^{m^j i^l}$ and $g^{\beta} = \prod_{j,l=0}^{f,t} (\mathbb{C}_{jl})^{i^j m^l}$.

verify-share($\mathbb{C}, m, \sigma$) verifies that σ is a valid share of P_m with respect to $\mathbb{C}$; it is true if and only if $g^{\sigma} = \prod_{j=0}^{t} (\mathbb{C}_{j0})^{m^j}$.

verify-shared($\mathbb{C}, Sig_C$) verifies the set of signatures Sig_C.

The parties may need to interpolate a polynomial a of degree f or a polynomial b of degree t. This can be done using standard Lagrange interpolation, we abbreviate this by saying a party *interpolates* a.

In the protocol description the variables e, f, and r count the number of echo, shared and ready messages. They are instantiated separately only for values of $\mathbb{C}$ that have actually been received in incoming messages.

Analysis. Proofs for the HAVSS properties mostly follow from [7] and for space limitation deferred to Appendix B.

3.4 HAVSS for Bootstrap of Hotstuff/SBFT

Although this paper focuses on fully asynchronous protocols, advancements in partially synchronous protocols [20, 35] have shown that the ability to generate distributively an $(f, 2f + 1)$-threshold key is a useful primitive. HAVSS is the first protocol that can power such efficient DKGs, for example, if we combine HAVSS with Hybrid-DKG [24] we can securely bootstrap Hotstuff and SBFT without introducing any new assumptions.

4 WEAK DISTRIBUTED KEY GENERATION

This section describes an asynchronous protocol for detecting agreement on the generation of (up to) $f + 1$ *candidate* shared keys without a trusted setup, which we use for building the eventually perfect coin in the next section. The key idea of wDKG is that the protocol never terminates (e.g., never commits to a specific key). Instead, each party outputs a finite sequence of candidate keys, and even though there is no explicit termination (otherwise, we would contradict the FLP [16] impossibility of asynchronous agreement), we guarantee that eventually all honest parties stop outputting new candidate keys and the last candidate key output by all honest parties is the same. Moreover, to bound the complexity of an higher-level protocol that uses our weak distributed key generation (wDKG), we guarantee that no honest party outputs more than $f + 1$ keys.

Algorithm 1 Protocol HAVSS for party P_i and tag $ID.d$ (sharing stage)

upon initialization **do**
  success $\leftarrow$ false
  for all $\mathbb{C}$ **do**
    $e_{\mathbb{C}} \leftarrow 0; r_{\mathbb{C}} \leftarrow 0$
    $A_{\mathbb{C}} \leftarrow \emptyset$; $B_{\mathbb{C}} \leftarrow \emptyset$ $Sig_{\mathbb{C}} \leftarrow \emptyset$

upon receiving "$ID.d$, in, share, s" **do** ▷ only P_d
  choose a random asymmetric bivariate polynomials u of degree (t, f) with $u(0, 0) = u_{00} = s$, i.e.,

$$u(x, y) = \sum_{j,l=0}^{t,f} u_{jl} x^j y^l$$

  $\mathbb{C} \leftarrow \{C_{jl}\}$, where $C_{jl} = g^{u_{jl}}$ for $j \in [0, t]$ and $l \in [0, f]$
  for $j \in [1, n]$ **do**
    $a_j(y) \leftarrow u(j, y)$; $b_j(x) \leftarrow u(x, j)$
    send "$ID.d$, send, $\mathbb{C}$, a_j, b_j" to P_j

upon receiving "$ID.d$, send, $\mathbb{C}$, a, b" from P_d for the first time **do**
  if verify – poly($\mathbb{C}$, i, a, b) **then**
    for $j \in [1, n]$ **do** send "$ID.d$, echo, $\mathbb{C}$, $a(j)$, $b(j)$" to P_j

upon receiving "$ID.d$, echo, $\mathbb{C}$, α, β" from P_m for the first time **do**
  if verify – point($\mathbb{C}$, i, m, α, β) **then**
    $A_{\mathbb{C}} \leftarrow A_{\mathbb{C}} \bigcup \{(m, \alpha)\}$; $B_{\mathbb{C}} \leftarrow B_{\mathbb{C}} \bigcup \{(m, \beta)\}$
    $e_{\mathbb{C}} \leftarrow e_{\mathbb{C}} + 1$
    if $e_{\mathbb{C}} = k$ **then**
      interpolate $\bar{a}$, $\bar{b}$ from $B_{\mathbb{C}}$, $A_{\mathbb{C}}$, respectively
      for $j \in [1, n]$ **do** send "$ID.d$, ready, $\mathbb{C}$, $\bar{a}(j)$, $\bar{b}(j)$, sig_i" to P_j

upon receiving "$ID.d$, ready, $\mathbb{C}$, α, β, sig_m" from P_m for the first time **do**
  if verify – point($\mathbb{C}$, i, m, α, β) **then**
    $Sig_{\mathbb{C}} \leftarrow Sig_{\mathbb{C}} \bigcup \{(m, sig_m)\}$
    $r_{\mathbb{C}} \leftarrow r_{\mathbb{C}} + 1$
    if $r_{\mathbb{C}} = n - f$ and $e_{\mathbb{C}} \geq k$ **then**
      $\bar{\mathbb{C}} \leftarrow \mathbb{C}$; $s_i \leftarrow \bar{a}(0)$; success $\leftarrow$ true
      for $j \in [1, n]$ **do** send "$ID.d$, shared, $\mathbb{C}$, $Sig_{\mathbb{C}}$, $\bar{b}(j)$" to P_j
      output ($ID.d$, out, shared)

upon receiving "$ID.d$, shared, $\mathbb{C}$, $Sig^m_{\mathbb{C}}$, β" from P_m for the first time **do**
  if verify – shared($\mathbb{C}$, $Sig^m_{\mathbb{C}}$) **then**
    if $e_{\mathbb{C}} \geq k$ **then** ▷Can fully terminate share
      $\bar{\mathbb{C}} \leftarrow \mathbb{C}$; $s_i \leftarrow \bar{a}(0)$; success $\leftarrow$ true
      for $j \in [1, n]$ **do** send "$ID.d$, shared, $\mathbb{C}$, $Sig_{\mathbb{C}}$, $\bar{b}(j)$" to P_j
      output ($ID.d$, out, shared)
    else if verify – point($\mathbb{C}$, i, m, β) **then** ▷Can only recover
      $B_{\mathbb{C}} \leftarrow B_{\mathbb{C}} \bigcup \{(m, \beta)\}$
      $r_{\mathbb{C}} \leftarrow r_{\mathbb{C}} + 1$
      if $r_{\mathbb{C}} = f + 1$ **then**
        $\bar{\mathbb{C}} \leftarrow \mathbb{C}$
        interpolate $\bar{a}$ from B_C,
        $s_i \leftarrow \bar{a}(0)$
        output ($ID.d$, out, shared)

Algorithm 2 Protocol HAVSS for party P_i and tag $ID.d$ (reconstruction stage)

1: **upon receiving** "$ID.d$, in, reconstruct" **do**
2: $c \leftarrow 0$; $S \leftarrow \emptyset$
3: **for** $j \in [1, n]$ **do** send "$ID.d$, reconstruct-share, s_i" to P_j

4: **upon receiving** "$ID.d$, reconstruct-share, σ" from P_m **do**
5: **if** verify − share($\bar{\mathbb{C}}$, m, σ) **then**
6: $S \leftarrow S \bigcup \{(m, \sigma)\}$; $c \leftarrow c + 1$
7: **if** $c = k$ **then**
8: interpolate a_0 from S
9: output ($ID.d$, out, reconstructed, $a_0(0)$)
10: **halt**

4.1 Definition

A weak Distributed Key Generation is a helper protocol that is implemented on top of n HAVSS instances where each party P_i acts as the dealer of HAVSS instance i. We denote the share that party P_i receives in HAVSS instance j by s_i^j, and define a *prediction* of a candidate distributed key to be a set of shares. During a wDKG each party P_i might output a sequence of predictions, and we say that an output prediction $\mathbb{P}_{ultimate}$ is *last* if P_i does not output a prediction after $\mathbb{P}_{ultimate}$. For each party P_i, there is a one-to-one mapping between a set of HAVSS dealers and the predictions induced by the HAVSS instances of these dealers. That is, given a set S of parties, the prediction $shares_i(S) \triangleq \{s_i^j \mid P_j \in S\}$, and given a prediction $\mathbb{P}$ of P_i, $source(\mathbb{P}) \triangleq \{P_j \mid s_i^j \in \mathbb{P}\}$. Note that $source(shares_i(S)) = S$. We say that two predictions $\mathbb{P}_1, \mathbb{P}_2$ of different parties are *matching* if $source(\mathbb{P}_1) = source(\mathbb{P}_2)$.

The wDKG protocol provides the following properties.

W(i): Inclusion. For every prediction $\mathbb{P}$ an honest party outputs, $|source(\mathbb{P})| \geq 2f + 1$.

W(ii): Containment. For each honest party P_i, predictions are ordered by strict containment. This means that for any two predictions output by P_i in times $k < j : \mathbb{P}_k \subset \mathbb{P}_j$.

W(iii): Eventual Agreement. Every honest party eventually outputs an ultimate prediction, and all ultimate predictions are matching.

W(iv): Privacy. If no honest party reveals its private share for a prediction p then the adversary can neither compute the prediction p nor the shared secret s. This is equivalent to the HAVSS privacy property defined before.

4.2 Technical Overview

The wDKG protocol uses n instances of HAVSS as sub-protocols. Each party P_i invokes HAVSS instance $ID.i$ as a dealer and participates in the sharing phases of all HAVSS instances as a receiver. Upon initialization, each party P_i instantiates its HAVSS with a random secret and collects $n - f$ shares from different HAVSS instances (including its own) into a prediction H. Note that since $n - f$ instances have honest leaders, then all honest parties eventually collect $n - f$ shares. Then, it starts the eventual agreement phase by broadcasting a candidate-key message that includes $source(H)$. Later, any time P_i delivers another HAVSS share, it inserts the share into H and broadcasts the new $source(H)$ in another candidate-key message.

When a party p_i receives $2f + 1$ candidate-key messages with the same source (set of parties) S, it (1) waits until $H \supseteq shares_i(S)$ or in other words until p_i gets all the HAVSS shares from instances with parties from S acting as dealers; and then (2) outputs the prediction $shares_i(S)$ provided it did not output a prediction $\mathbb{P} \not\subset shares_i(S)$ before to make sure

parties output increasing predictions by containment. Note that by the containment property and since predictions by honest parties consists of at least $2f + 1$ shares, we get that each party outputs at most $f + 1$ predictions.

Although the above protocol has an efficient ($O(n^4)$) as we will see later) word complexity, it needs one further check in order to avoid exponential computation and storage. The challenge is that every time a party receives a new candidate-key it needs to search its local memory to increase the counter of how many matching $source(H)$ it has received. Honest parties broadcast up to $f + 1$ candidate-key messages, but a Byzantine party might broadcast an exponential number of such messages, causing the local memory and the cost of searching it to become exponential. Therefore, in order to avoid this attack we ignore candidate-key messages from parties that do not satisfy containment (i.e., a party p_i ignores a candidate-key message with source set S from party p_j if it previously received from p_j a candidate-key message with source set $S' \not\subset S$). The pseudocode appears in Algorithm 3.

Algorithm 3 Protocol wDKG for party P_i

```
1:  upon initialization do
2:      for every j ∈ {1, ..., n} do
3:          S_j ← {}                          ▷The source (set of parties) p_i received from p_j
4:      H ← {}                                ▷ The set of HAVSS shares p_i outputs
5:      S_ℙ ← {}                              ▷The source set of the current prediction
6:      C[:] ← 0                              ▷A counter for every possible source
7:      select random r_i
8:      invoke (i, in, share, r_i)            ▷Every party starts an HAVSS as a dealer

9:  upon (ID.j, out, shared) do
10:     H ← H ∪ {s_i^j}
11:     if |H| ≥ n − f then
12:         send "candidate-key, source(H)" to all parties

13: upon receiving "candidate-key, S" from party p_j do   ▷Handle these messages one after the other
14:     if S ⊃ S_j ∪ S_ℙ then
15:         S_j ← S
16:         C[S] ← C[S] + 1
17:         if C[S] = n − f then
18:             S_ℙ ← S
19:             wait until H ⊇ shares_i(S)
20:             output (out, key, shares_i(S))
```

4.3 Analysis

In this Section we prove that the protocol in Figure 3 implements wDKG. The first two proofs follow directly from the code. For the eventual agreement we first need to show that no honest party will get stuck at a prediction that is not the best possible. Then we show that parties will keep delivering predictions that include more shares until they deliver a prediction with the maximum number of shares (all the shares that were generated by good dealers). Since no party gets stuck at a suboptimal prediction and there exists a maximum prediction, all parties will eventually deliver that prediction and stop delivering anything new, hence they eventually agree. Of course the parties will not be aware that the prediction they delivered is the maximum, which is the reason they cannot explicitly terminate. Specifically, we prove the following lemmas:

4.3.1 Correctness proof. In this section we prove that the protocol in Figure 3 implements wDKG, i.e., satisfies *containment, inclusion, and eventual agreement*:

Lemma 4.1. *The protocol in Algorithm 3 satisfies W(ii) (Containment).*

Proof. By line 14, honest parties ignore "candidate-key, S" messages when $S \not\supset S_{\mathbb{P}}$. By the code, $S_{\mathbb{P}}$ stores the source set of the last prediction. The lemma follows from the fact that candidate-key messages never handled in parallel.

□

Lemma 4.2. *The protocol in Algorithm 3 satisfies W(i) (Inclusion).*

Proof. Let $\mathbb{P}$ be a prediction some honest party P_i outputs. By line 17, P_i gets at least $n-f$ "candidate-key, *source*($\mathbb{P}$)" messages. Thus, at least one honest party sends a "candidate-key, *source*($\mathbb{P}$)" message. Therefore, by line 11, $|\textit{source}(\mathbb{P})| = |\mathbb{P}| \geq n - f$.

□

Lemma 4.3. *An honest party is never stuck.*

Proof. The only possible place for an honest party to stuck is in Line 19. Consider an honest party P_i that gets to Line 19 and waits until its $H \supseteq \textit{shares}_i(S)$ where S is the source set it received in the candidate-key message. By Line 17, P_i gets $n - f$ "candidate-key, S" messages, and thus at least one honest party P_j sent "candidate-key, S" message. By the code, P_j delivers a share for every HAVSS instance in S. Thus, by property H(ii), P_i will eventually deliver a share for every HAVSS instance in S as well. Meaning that eventually $H \supseteq \textit{shares}_i(S)$, and thus P_i will eventually end the waiting in Line 19.

□

Lemma 4.4. *The protocol in Algorithm 3 satisfies W(iii) (Eventual Agreement).*

Proof. Note that the size of H is bounded by n, so for every honest party there is a point after which H is never changing and includes all HAVSS shares it will ever deliver. By H(ii), all honest parties will eventually reach the same *source*(H), which we denote by S_H.

We now show that an honest party P_i does not ignore a "candidate-key, S_H" message from an honest party P_j. In other words, the if statement in Line 14 is always true when P_j receives such message. We need to show two conditions:

- First, $S_H \supset S_j$. Since by the code, P_j only sends the "candidate-key$''$ with *source*(H), we get by the definition of S_H that P_j never sends "candidate-key, S'" message with $S' \not\subseteq S_H$.
- Second, $S_H \supset S_{\mathbb{P}}$. Assume by a way of contradiction that at some point P_i sets $S_{\mathbb{P}} \leftarrow S'$ s.t. $S' \not\subseteq S_H$. By the code, P_i gets "candidate-key, S'" message from at least one honest party P_k. Therefore, the *source*(H) of party P_j was equal to S' at some point. A contradiction to the definition of S_H.

By property H(i), and since we have at least $n - f$ honest parties, we get that $|S_H| \geq n - f$. Thus, by the code, all honest parties will eventually send "candidate-key, S_H" message to all other honest parties. Therefore, by Lemma 4.3 and from the above, every honest party P_i will eventually process $n - f$ "candidate-key, S_H" message, pass the if statement in Line 17, and output $\textit{shares}_i(S_H)$.

It is left to show that no honest party will ever output a prediction after $shares_i(S_H)$. Assume by a way of contradiction that some party P_i outputs S' after it outputs $shares_i(S_H)$. By property W(ii) (Containment), $S' \supset S_H$. Thus, by definition of S_H, S' contains a party that acts as a dealer in a HAVSS instance in which no honest party delivers a share. Therefore, no honest party ever sends a "candidate-key, S'" message. Hence, P_i never get $n - f$ "candidate-key, S'" messages, and thus by the code never output S'. A contradiction.

□

Lemma 4.5. *The protocol in Algorithm 3 satisfies W(iv) (Privacy).*

Proof. Follows directly from the W(i) (inclusion) and H(iv) (privacy).

□

Complexity. By the code, each party sends at most $f + 1$ candidate-key messages, each of which of size $O(n)$, to all other parties. Therefore, the bit complexity of each party is $O(n^3)$ words, and the total bit complexity is $O(n^4)$ words.

5 FROM WEAK DKG TO EVENTUALLY PERFECT COMMON COIN

In this section, we use wDKG as the backbone of an *eventually-perfect common coin (EPCC)*, which is a perfect-common coin that fails a finite number of times (at most f in our case). As a result, we can use it as a perfect-coin as long as we make sure to handle the small number of disagreements.

5.1 Definition

The EPCC is a long-lived task, which can be invoked many times by each party via coin-toss(sq) invocation. Each invocation is associated with a unique sequence number sq and returns a value v. We assume well-formed executions in which honest parties block any subsequent EPCC invocations until the invoked EPCC returns a value. This is crucial for the Eventual Agreement property because disagreement on the EPCC output in one instance must advance at least one wDKG key toward the following instance. For notational convenience, we assume that if a party invokes coin-toss(sq) and later invoke coin-toss(sq'), then $sq' > sq$.

An EPCC implementation must satisfy the following properties:

E(i): Unpredictability. For every sq, the probability that the adversary predicts the return value of coin-toss(sq) invocation by an honest party before at least one honest party invoke coin-toss(sq) is at most $1/2 + \epsilon(k)$, where $\epsilon(k)$ is a negligible function.

E(ii): Termination: If $n - f$ honest parties invoke coin-toss(sq), then all coin-toss(sq) invocations by honest parties eventually return.

E(iii): Eventual Agreement: There are at most f sequence numbers sq for which two invocations of coin-toss(sq) by honest parties return different coins.

5.2 Technical Overview

Our EPCC protocol is built on top of n HAVSS instances and uses the wDKG algorithm as a sub-protocol. Recall that the wDKG algorithm outputs a sequence of at most $f + 1$ predictions (sets of HAVSS shares) $\mathbb{P}_1, \ldots, \mathbb{P}_l$. Whenever, the wDKG sub-protocol outputs a prediction $\mathbb{P}_i$ we use it to derive a tuple $\langle K_{\mathbb{P}_i}, V_{\mathbb{P}_i} \rangle$, where $K_{\mathbb{P}_i}$ is the *key*, and $V_{\mathbb{P}_i}$ is the a *bit vector* indicating the HAVSS instances included in $source(\mathbb{P}_i)$ (see get-key below). The $\langle K, V \rangle$ variables store the last derived key, and the bit vector, respectively, and are updated whenever the wDKG outputs a new prediction.

Upon a coin-toss(sq) invocation by an honest party P_i, it enters a protocol to construct a common coin. The protocol loops using the outputs from wDKG until for some key K, P_i succeeds in collecting $n - f$ shares corresponding to K and the sequence number sq. More specifically, each party P_i uses the latest key K, V output from wDKG and the sequence number sq to generate its share of the common-coin, and sends a coin-share message with the share together with the bit vector V to all other parties. Whenever the wDKG outputs a new prediction, P_i updates the $\langle K, V \rangle$ variables, and broadcasts a new share.

A coin-toss(sq) invocation by an honest party P_i returns when it collects $2f + 1$ coin-share messages from different parties with valid coin-shares and the same bit vector V'. Note that V' can be different from any bit vector party P_i previously sent in a coin-share message. To validate the coin-shares, P_i needs to generate a *commitment* $C_{v'}$ that is associated to the bit vector V' by combining all the commitments of HAVSS instances included in V' (see get-commitment below). Note that in order to be able to do it, P_i first needs to complete the sharing phases of all HAVSS instances included in V'. Then, after P_i successfully verifies the $2f + 1$ signatures (see verify-share below), it uses them to produce a coin (see generate-coin below), sends it in a coin message together with the bit vector to all other parties, and outputs it.

Upon receiving a coin message, P_i first checks that the bit vector includes at least $2f + 1$ ones in order make sure randomness from honest parties were included in the associated key generations. Next, P_i generates a *commitment* associated with the bit vector and then uses it to verify the coin (see verify-coin below). If the verification passes, P_i forwards the coin message to all parties and output the coin.

Note that since EPCC is a long lived object some honest parties may complete a coin-toss(sq) for some sq before another honest party invoked coin-toss(sq). To handle this, honest parties maintain two maps S and *Coins* that map tuples of bit-vectors and sq to set of coin-shares and coins, respectively. These maps are updated every time a share-coin or coin message is received regardless if there is a coin-toss(sq) operation in progress. In addition, when a coin-toss(sq) operation invoked by an honest party P_i, it first checks these maps to see if it already received enough messages to return a coin. The pseudocode is given below (Algorithm 4) and the omitted proofs are given in Appendix C. In the pseudocode we use the following functions:

get-key($\mathbb{P}$) gets a prediction output $\mathbb{P}$ from a wDKG sub-protocol, and outputs $\langle K_{\mathbb{P}}, V_{\mathbb{P}} \rangle$ that are computed as follows:

$$K_{\mathbb{P}} = \sum_{s \in \mathbb{P}} s \quad \text{and} \quad \forall p_i \in \mathit{source}(\mathbb{P}), V_{\mathbb{P}}[i] = 1$$

In other words, $K_{\mathbb{P}}$ is the sum of all shares in $\mathbb{P}$ and $V_{\mathbb{P}}$ indicates the HAVSS instances these shares came from.

get-commitment($V_{\mathbb{P}}$) gets a bit vector that was generated from a prediction $\mathbb{P}$, and returns a commitment $C_{\mathbb{P}}$ that is used to verify signatures associated with $K_{\mathbb{P}}$ (share and coin). In order to be able to compute $C_{\mathbb{P}}$, parties first have to complete the sharing phases of all the HAVSS instance indicated by $V_{\mathbb{P}}$ in order to get their commitment, and then multiply them to get $C_{\mathbb{P}}$. More specifically,

$$\forall i \in \{1, \ldots, n\}, \text{if } V_{\mathbb{P}}[i] = 1, \text{then wait for commitment } C_i$$

$$\text{from } P_i's \text{ HAVSS instance}$$

$$C_{\mathbb{P}} = \prod_{i=1}^{n} V_{\mathbb{P}}[i] C_i$$

In Algorithm 4, an invocation of get-commitment can block forever if send by a bad party that lies about what HAVSS instances have terminated. We do not need to handle this as we only care to return one random value of a *sq*. To this end, we handle all events concurrently and abort all outstanding procedures associated with *sq* after we output a coin for *sq*.

Note that for every prediction $\mathbb{P}$ an honest party gets from the wDKG protocol, the bit vector $V_{\mathbb{P}}$ defines a unique private $K_{\mathbb{P}}^{i}$ for every party P_i, and a unique global commitment $C_{\mathbb{P}}$. Together, they form the setup required for the Diffie-Hellman based threshold coin-tossing scheme that is given in [9], which yields a common coin flip for each *sq* input. In our educational example, we use Pedersen [32] DKG, which does not produce uniformly random keys [19], but as shown by Libert et al. [28] it is sufficient for the adaptively secure threshold signatures, which we will use for the real-world deployment. Hence we assume that the key generated by the DKG is sufficiently random for our proofs and only focus on proving that it remains unpredictable and private. Below we briefly describe the functionality this scheme provides, and more details and formal proofs can be found in [9]. Note that the wDKG might output different sequences of predictions when invoked by different parties, so the challenge that we overcome in Algorithm 4 is how to eventually agree on the same key.

generate-share($C_{\mathbb{P}}, K_{\mathbb{P}}, sq$) uses the key $K_{\mathbb{P}}$ derived from prediction $\mathbb{P}$ to sign the sequence number *sq* in order to generate a share for a coin defined by $C_{\mathbb{P}}$ and *sq*.

verify-share($C_{\mathbb{P}}, sq, j, \sigma$), verifies that the given value σ is a valid coin share from P_j for the coin defined by $C_{\mathbb{P}}$ and *sq*.

generate-coin($C_{\mathbb{P}}, \Sigma, sq$) uses a set Σ of $2f + 1$ valid shares defined by $C_{\mathbb{P}}$ and *sq* in order to generates the coin.

verify-coin($C_{\mathbb{P}}, \sigma, sq$), verifies that the given value σ is a valid coin defined by $C_{\mathbb{P}}$ and *sq*.

5.3 Analysis

5.3.1 Correctness proof. In this section we show *unpredictability, termination, and eventual agreement* of our EPCC. The first two properties can easily be deduced from the code. For eventual agreement we first need to show that (due to WDKG's containment property) if a party uses a certain set of shares V_1 to produce randomness then it will only use supersets of V_1 in future invocations. This creates a total ordering of predictions. The second part of the proof relies on the well-formed nature of EPCC and shows that if different sets of shares where used to generate randomness for a certain invocation *sq* then only the largest set of shares will be used for any subsequent invocation. Given that there can only be $f + 1$ different valid and totally ordered sets, the adversary can only cause the generation of inconsistent randomness at most f times. Specifically, we prove in Appendix C the following Lemmas:

Lemma 5.1. *If a valid coin for some sq is generated, then at least* $2f + 1$ *valid share-coins associated with some bit vector* V *for sq were previously generated,* $f + 1$ *of which by honest parties.*

Lemma 5.2. *The protocol in Algorithm 4 satisfies E(i) (Unpredictability).*

Lemma 5.3. *For every sq, if an invocation of* coin-toss(sq) *by an honest party* P_i *returns, then all* coin-toss(sq) *invocations by honest parties eventually return.*

Lemma 5.4. *The protocol in Algorithm 4 satisfies E(ii) (Termination).*

Algorithm 4 Protocol EPCC for party P_i. All events must be handled in parallel per sq. Upon first output message for sq all other invocations are aborted.

upon initialization **do**
  invoke wDKG
  $K \leftarrow \perp$; $V \leftarrow \perp$ ▷last derived key and bit vector, respectively
  $currentSQ \leftarrow \perp$ ▷$\perp$ indicates that there is not coin-toss in progress
  $S[:] \leftarrow \{\}$ ▷A mapping from tuples of bit vector and sq to sets of shares
  $Coins[:] \leftarrow \perp$ ▷A mapping from tuples of bit vector and sq to coins

upon (out, key, $\mathbb{P}$) **do** ▷prediction output form the wDKG sub-protocol
  $\langle K, V \rangle \leftarrow$ get-key($\mathbb{P}$)
  if $currentSQ \neq \perp$ **then**
    BroadcastShare()

upon coin-toss(sq) **do**
  $currentSQ \leftarrow sq$ ▷Avoid races during concurrent invocations
  if $\exists V'$ s.t. $Coins[\langle V', sq \rangle] \neq \perp$ **then** ▷Already saw the coin
    ForwardCoinAndReturn(V', sq)
  if $\exists V'$ s.t. $|S[\langle V', sq \rangle]| \geq 2f + 1$ **then** ▷Enough shares
    BroadcastCoinAndReturn(V', sq)
  if $V \neq \perp$ **then**
    BroadcastShare()

upon receiving "coin-share, sq, σ, V_j" message from party P_j for the first time **do**
  $C \leftarrow$ get-commitment(V_j)
  if verify-share(C, sq, j, σ) $\wedge$ $\sum_{k=1}^{n} V_j[k] \geq 2f + 1$ **then**
    $S[\langle V_j, sq \rangle] \leftarrow S[\langle V_j, sq \rangle] \cup \{\sigma\}$
    if $sq = currentSQ \wedge |S[\langle V_j, sq \rangle]| \geq 2f + 1$ **then**
      BroadcastCoinAndReturn(V_j, sq)

upon receiving "coin, sq, ρ, V_j" message from party P_j for the first time **do**
  $C \leftarrow$ get-commitment(V_j)
  if verify-coin(C, ρ, sq) $\wedge$ $\sum_{k=1}^{n} V_j[k] \geq 2f + 1$ **then**
    $Coins[\langle V_j, sq \rangle] \leftarrow \rho$
    if $sq = currentSQ$ **then**
      ForwardCoinAndReturn(V_j, sq)

procedure BroadcastShare()
  $C \leftarrow$ get-commitment(V)
  $\sigma \leftarrow$ generate-share(C, K, $currentSQ$)
  send "coin-share, $currentSQ$, σ, V" to all parties

procedure BroadcastCoinAndReturn(V', sq)
  $C \leftarrow$ get-commitment(V')
  $\rho \leftarrow$ generate-coin(C, $S[\langle V', sq \rangle]$, sq)
  send "coin, sq, ρ, V'" to all parties
  $currentSQ \leftarrow \perp$
  output (out, coin, sq, ρ)

procedure ForwardCoinAndReturn(V', sq)
  send "coin, sq, $Coins[\langle V', sq \rangle]$, V'" to all parties
  $currentSQ \leftarrow \perp$
  output (out, coin, sq, $Coins[\langle V', sq \rangle]$)

Lemma 5.5. *If an honest party generates a share-coin associated with V, then it will never generate a share-coin associated with $V' \not\supseteq V$.*

Lemma 5.6. *If for some sq, two* coin-toss(sq) *invocations by two honest parties return different valid coins $\rho_1 \neq \rho_2$, then there are two bit vectors V_1, V_2 s.t. (1) $V_1 \subset V_2$; and (2) $f+1$ honest parties generated valid share-coins associated with V_1 for sq and $f+1$ honest parties generated valid share-coins associated with V_2 for sq.*

Lemma 5.7. *For every $1 \leq k \leq f+1$, if there are k sequence numbers sq for which two invocations of* coin-toss(sq) *by honest parties output different coins, then there is a bit vector V of size at least $2f+1+k$ such that $f+1$ honest parties generated valid share-coins associated with V.*

Lemma 5.8. *The protocol in Algorithm 4 satisfies E(iii) (Eventual Agreement).*

5.3.2 Complexity. By W(i) and W(ii), each party outputs at most $f+1$ predictions from the wDKG sub-protocol. For each predictions, each party sends at most a constant number of words and $O(n)$ sized bit-vector to every party. Hence the worst-case complexity of a consistent coin flipping is $O(n^4)$ bits + $O(n^3)$ words.

6 ACHIEVING CONSENSUS

6.1 Eventually Efficient Asynchronous Binary Agreement

Once we have our EPCC, we can use it in any Binary Agreement protocol that uses a weak coin [6, 29]. The most efficient asynchronous BA solution is from Moustefaoui's et al [29] and has $O(n^2)$ bit complexity [5].

Since our coin has at most f bad flips, when we plug it in [29] we know that if we invoke n instances of ABA in succession with the same coin, then the overall number of bad flips remains f in the entire succession. Hence, the overall complexity remains $O(n^3)$ bit complexity and expected $O(f)$ rounds. We refer to an ABA that has this succession property as eventually efficient ABA (EEABA).

We refrain from reintroducing the full protocol as we only need to plug in our coin-toss(sq) and make sure that a party which has already seen a safe value continues to coin-toss(sq) in order for EPCC to be live, but ignores the output of EPCC (as it already knows the safe value). The total bit complexity of our EEABA has two parts. First, there is the needed HAVSS for EPCC to work, which has a total $O(n^4)$ words (n concurrent instances of HAVSS). Then, we can start running the ABA of [29] which (as mentioned above) has an overall complexity remains $O(n^3)$ bit complexity and expected $O(f)$ rounds. Hence the total complexity of EEABA is $O(n^4)$ bit complexity and expected $O(f)$ rounds. Nevertheless, if we run this protocol for ($O(n^2)$) sequential decisions it will amortize to $O(n^2)$ communication complexity and $O(1)$ termination because the coin will be perfect for most of the EEABA instances (at most f failures due to asynchrony) which means that the $n^2 - f$ instances will terminate in an expected number of 2 rounds. Hence, we can get the ABA with the properties defined in Lemma 1.3.

6.2 Asynchronous Distributed Key Generation

We build our ADKG protocol on top of EEABA by explicitly terminating the wDKG and agreeing on what HAVSS instances contribute to the scheme. We can achieve this by extending the *Asynchronous Common Subset (ACS)* protocol introduced by Ben-or et al [5]. In an ACS protocol, n processors have some initial value and they need to agree on a

[5] They do not give an implementation for their weak coin assumption, but instead use an external oracle.

Algorithm 5 Protocol ADKG for party P_i

upon initialization **do**
  $I \leftarrow \Pi$ ▷A set of parties, initially all
  $K \leftarrow \{\}$ ▷The set of HAVSS shares that corresponds the the agreed instances
  $c \leftarrow 0$ ▷A counter for the number of ABAs in which P_i decided
  select random r_i
  invoke (i, in, share, r_i) ▷Every party starts an HAVSS as a dealer

upon ($ID.j$, out, shared) **do** ▷The sharing phase of P_j's HAVSS completed
  if $P_j \in I$ **then**
    invoke $ABA.j$ with 1
    $I \leftarrow I \setminus \{P_j\}$ ▷Remove instances already voted on

upon ($ABA.j$, deliver, 1) **do**
  $K \leftarrow K \cup \{s_i^j\}$ ▷This might block until the HAVSS delivers, but it will eventually terminate.
  $c \leftarrow c + 1$
  if $c = n - f$ **then**
    for all $P_l \in I$ **do**
      invoke $ABA.l$ with 0
      $I \leftarrow I \setminus \{P_l\}$ ▷Remove instances already voted on
  if $c = n$ **then**
    output K

upon ($ABA.j$, deliver, 0) **do**
  $c \leftarrow c + 1$
  if $c = n$ **then**
    output K

subset of values to be adopted. Our Asynchronous Distributed Key Generation is similar, with the added restriction that the values we agree on need to remain private (secret-shared), hence parties output the same set of parties *source(v)* and maintain a private shares set v locally. For simplicity, we do not deal in this section with the specific details of how to a generate a secret-key, public-key, and the commitments for verification, which is fairly straightforward after we agree on the set of HAVSS instances.

6.2.1 Definition. More formally, an Asynchronous Distributed Key Generation protocol is a one-shot consensus variant. Each party is initialized with an $ID.i$ of the HAVSS instance it should act as a dealer, as well as the full ID vector of the HAVSS instances it should be a part of. For every party p_i, the protocol outputs a private set of shares v_i s.t. the following is satisfied except with negligible probability:

A(i): Validity. If an honest party outputs a set of shares v, then $|v| \geq n - f$ and v includes only valid shares.

A(ii): Agreement. For every two honest parties P_i, P_j, if P_i and P_j output sets of shares v_i and v_j, respectively, then $source(v_i) = source(v_j)$.

A(iii): Liveness. If $n - f$ correct parties start dealing shares and the adversary delivers all messages, then all correct parties output a set of shares.

A(iv): Privacy. If an honest party p_i outputs a set of shares v_i and no honest party has revealed its output shares and the secret it shared, then the adversary cannot compute the sum of secrets shared by parties in $source(v_i)$.

6.2.2 Technical Overview. We follow the ACS solution of Ben-Or et al [5], which consists of starting n parallel reliable broadcasts, one for each party to act as the sender, where for each broadcast instance, they use a single ABA to agree whether its value should be included in the set. In their protocol, parties invoke with 1 (success) every ABA that corresponds to a reliable broadcast instance in which they deliver a value, and refraining from invoking with 0 any ABA instance until $n - f$ ABA instances have decided 1. Then, they invoke with 0 all other ABA instance and terminate the ACS protocol once they decided in all ABA instances.

Our ADKG protocol is similar but instead of reliable broadcasts, we uses HAVSS instances. By the agreement and liveness properties of the HAVSS, eventually there are $n - f$ ABA instances which all honest parties invoke with 1 and thus eventually $n - f$ instances agree on 1 (all honest parties decide 1). Note that the properties of the binary ABA guarantee that if all honest parties invoke it with 1, then they all eventually decide 1 (same for 0). This protocol has an expected running time of $O(log(n))$. Additionally EEABA has an expected running time of $O(1)$ when the network is synchronized and an expected running time of $O(f)$ when the adversary is manipulating the message ordering hence the full ADKG protocol has an expected $O(logn)$ running time without a network level adversary and an $O(f + logn) = O(f)$ running time under asynchrony. On a high-level the ADKG works as follows:

When a party is initialized for ADKG it also initializes n parallel ABA instances of Section 6.1 s.t. $ABA.j$ will be used to decide if HAVSS $ID.j$ terminated successfully (all honest parties delivered a share that corresponds to the same secret), and proceeds as follows:

(1) Once player P_i delivers an HAVSS share for P_j's instance he inputs 1 in $ABA.j$.
(2) Once P_i decides 1 in **n-f** ABA instances, it inputs 0 in every ABA instance it have not invoked yet.
(3) When P_i decides in all **n** ABA instances, p_i outputs the subset K of shares that corresponds to ABA instance in which it decided 1.

A detailed description of the protocol is given in Algorithm 5 and the proof is given in Appendix D.

Analysis. The cost of n parallel instances (where each instance costs a worst case of $O(n^3)$ and has an expected $O(n)$ running time) is $O(n^4)$ the same as the HAVSS step. Once the ADKG terminates the system can use the strong common-coin generated to run VABA [1] and amortize the costs to $O(n^2)$. We know that *validity, agreement* and *liveness* hold from ACS. Privacy holds from *inclusion* and *privacy* of the wDKG. With this we prove our main Theorem.

7 CONCLUSION

In this paper, we show a protocol that implements the first asynchronous Distributed Key Generation protocol. To achieve this we show how to get the first AVSS protocol that supports thresholds $f + 1 < k \leq 2f + 1$, the first Eventually Efficient ABA which does not need a trusted setup and can also be amortized to the optimal cost if run $O(n^2)$ times in sequence, and the first VABA that does not require a trusted setup.

ACKNOWLEDGEMENTS

We would like to thank Ittai Abraham for the discussions and guidance during the initial conception of the project, especially for HAVSS. Furthermore, we would like to thank the anonymous reviewers for pointing out the relevance of this work to MPC protocols.

REFERENCES

[1] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

[2] Abhinav Aggarwal, Mahnush Movahedi, Jared Saia, and Mahdi Zamani. Bootstrapping public blockchains without a trusted setup. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 366–368, 2019.

[3] Georgia Avarikioti, Eleftherios Kokoris Kogias, and Roger Wattenhofer. Brick: Asynchronous state channels. *arXiv preprint arXiv:1905.11360*, 2019.

[4] Laasya Bangalore, Ashish Choudhury, and Arpita Patra. Almost-surely terminating asynchronous byzantine agreement revisited. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 295–304. ACM, 2018.

[5] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 183–192. ACM, 1994.

[6] Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 154–162. ACM, 1984.

[7] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 88–97. ACM, 2002.

[8] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.

[9] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[10] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 191–201. IEEE, 2005.

[11] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC*, volume 93, pages 42–51. Citeseer, 1993.

[12] Ashish Choudhury. Optimally-resilient unconditionally-secure asynchronous multi-party computation revisited. Cryptology ePrint Archive, Report 2020/906, 2020. https://eprint.iacr.org/2020/906.

[13] Ashish Choudhury and Arpita Patra. An efficient framework for unconditionally secure multiparty computation. *IEEE Transactions on Information Theory*, 63(1):428–468, 2016.

[14] Ran Cohen. Asynchronous secure multiparty computation in constant time. In *Public-Key Cryptography–PKC 2016*, pages 183–207. Springer, 2016.

[15] Sandro Coretti, Juan Garay, Martin Hirt, and Vassilis Zikas. Constant-round asynchronous multi-party computation based on one-way functions. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 998–1021. Springer, 2016.

[16] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 1985.

[17] Bryan Ford, Philipp Jovanovic, and Ewa Syta. Que sera consensus: Simple asynchronous agreement with private coins and threshold logical clocks. *arXiv preprint arXiv:2003.02291*, 2020.

[18] Juan A Garay, Aggelos Kiayias, Nikos Leonardos, and Giorgos Panagiotakos. Bootstrapping the blockchain, with applications to consensus and fast pki setup. In *IACR International Workshop on Public Key Cryptography*, pages 465–495. Springer, 2018.

[19] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 295–310. Springer, 1999.

[20] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 568–580. IEEE, 2019.

[21] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

[22] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Cryptographic asynchronous multi-party computation with optimal resilience. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 322–340. Springer, 2005.

[23] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Asynchronous multi-party computation with quadratic communication. In *International Colloquium on Automata, Languages, and Programming*, pages 473–485. Springer, 2008.

[24] Aniket Kate, Yizhou Huang, and Ian Goldberg. Distributed key generation in the wild. *IACR Cryptology ePrint Archive*, 2012:377, 2012.

[25] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th {usenix} security symposium ({usenix} security 16)*, pages 279–296, 2016.

[26] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Sandra Deepthy Siby, Nicolas Gailly, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. Calypso: Auditable sharing of private data over blockchains. *IACR Cryptol. ePrint Arch., Tech. Rep*, 209:2018, 2018.

[27] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[28] Benoît Libert, Marc Joye, and Moti Yung. Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. *Theoretical Computer Science*, 645:1–24, 2016.

[29] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with t< n/3, o (n2) messages, and o (1) expected time. *Journal of the ACM (JACM)*, 62(4):31, 2015.

[30] Arpita Patra, Ashish Choudhary, and C Pandu Rangan. Efficient statistical asynchronous verifiable secret sharing with optimal resilience. In *International Conference on Information Theoretic Security*, pages 74–92. Springer, 2009.
[31] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.
[32] Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 522–526. Springer, 1991.
[33] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.
[34] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 444–460. Ieee, 2017.
[35] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

A FULL COMPUTATIONAL MODEL

Following [1, 8, 9], we use standard modern cryptographic assumptions and definitions. We model the computations made by all system components as probabilistic Turing machines, and bound the number of computational basic steps allowed by the adversary by a polynomial in a *security parameter k*. A function $\epsilon(k)$ is *negligible* in k if for all $c > 0$ there exists a k_0 s.t. $\epsilon(k) < 1/k^c$ for all $k > k_0$. A computational problem is called *infeasible* if any polynomial time probabilistic algorithm solves it only with negligible probability. Note that by the definition of infeasible problems, the probability to solve at least one such problem out of a polynomial in k number of problems is negligible. Intuitively, this means that for any protocol P that uses a polynomial in k number of infeasible problems, if P is correct provided that the adversary does not solve one of its infeasible problems, then the protocol is correct except with negligible probability. We assume that the number of parties n is bounded by a polynomial in k.

Communication. We assume asynchronous links controlled by the adversary, that is, the adversary can see all messages and decide when and what messages to deliver. In order to fit the communication model with the computational assumptions, we restrict the adversary to perform no more than a polynomial in k number of computation steps between the time a message m from an honest party p_i is sent to an honest party p_j and the time m is delivered by p_j[6]. In addition, for simplicity, we assume that messages are *authenticated* in a sense that if an honest party p_i receives a message m indicating that m was sent by an honest party p_j, then m was indeed generated by p_j and sent to p_i at some prior time. This assumption is reasonable since it can be easily implemented with standard symmetric-key cryptographic techniques in our model.

Termination. Note that the traditional definition of the liveness property in distributed system, which requires that all correct (honest) parties *eventually* terminate provided that all messages between correct (honest) parties eventually arrive, does not make sense in this model. This is because the traditional definition allows the following:

- Unbounded delivery time between honest parties, which potentially gives the adversary unbounded time to solve infeasible problems.
- Unbounded runs that potentially may consist of an unbounded number of infeasible problems, and thus the probability that the adversary manages to solve one is not negligible.

Following Cachin et al. [8, 9], we address the first concern by restricting the number of computation steps the adversary makes during message transmission among honest parties. So as long as the total number of messages in the protocol is polynomial in k, the error probability remains negligible. To deal with the second concern, we do not use a standard

[6]Note that although this restriction gives some upper bound on the communication in terms of the adversary local speed, the model is still asynchronous since speeds of different parties are completely unrelated.

liveness property in this paper, but instead we reason about the total number of messages required for all honest parties to terminate. We adopt the following definition from [8, 9]:

Definition A.1 (Uniformly Bounded Statistic). Let X be a random variable. We say that X is *probabilistically uniformly bounded* if there exist a fixed polynomial $T(k)$ and a fixed negligible functions $\delta(l)$ and $\epsilon(k)$ such that for all $l, k \geq 0$,

$$Pr[X > lT(k)] \leq \delta(l) + \epsilon(k)$$

With the above definition Cachin et al. [8, 9] define a progress property that makes sense in the cryptographic settings:

- *Efficiency: The number of messages generated by the honest parties is probabilistically uniformly bounded*

The efficiency property implies that the probability of the adversary to solve an infeasible problem is negligible, which makes it possible to reason about the correctness of the primitives' properties. However, note that this property can be trivially satisfied by a protocol that never terminates but also never sends any messages. Therefore, in order for a primitive to be meaningful in this model, Cachin et al. [8, 9] require another property:

- *Termination*: If all messages sent by honest parties have been delivered, then all honest parties terminated.

B HAVSS PROOFS

Lemma B.1. *The protocol in Algorithms 1 and 2 satisfy H(i) (Liveness).*

Proof. If the dealer p_d is honest, it follows directly by inspection of the protocol that all honest parties complete the sharing $ID.d$, provided all parties initialize the sharing $ID.d$ and the adversary delivers all associated messages.

□

Lemma B.2. *The protocol in Algorithms 1 and 2 satisfy H(ii) (Agreement).*

Proof. We show that if some honest party p_i completes the sharing of $ID.d$, then all honest parties will complete the sharing of $ID.d$, provided all parties initialize the sharing $ID.d$ and the adversary delivers all associated messages. Consider two cases:

- First, p_i completes the sharing directly (line 29 or line 35 in Algorithm 1). Then it has received $n - f$ valid ready messages that agree on some $\bar{\mathbb{C}}$ from a set of at least $n - f$ parties S. Since we have at most f Byzantine parties, we get that S contains at least $n - 2f$ honest parties who have witnessed k valid echo messages and thus each such party will also complete the sharing upon reception of $n - f$ ready messages. By the algorithm in step 4 (line 28 or 34), after receiving $n - f$ (signed) valid ready messages, p_i sends them to all other parties. Therefore, every honest party in S eventually receives $n - f$ valid ready messages and thus eventually outputs shared. It is left to show that honest parties not in S will terminate as well. Consider such party p_j that never sent a ready message. We already showed that eventually $f + 1$ honest parties in S output shared, which means that they had a correct $b(j)$ polynomial and they will eventually sent a shared message with a valid point to p_j. Therefore, p_j eventually gets at least $f + 1$ consistent shared messages, recovers its share in step 5 and terminates as well (line 36-43).
- Second, p_i complete the sharing indirectly (line 36-43). Then we know that p_i gets at least $f + 1$ consistent shared messages, meaning that p_i gets at least one such message from an honest party p_j. By step 4, p_j was part of S and terminated (line 29 or 35). Therefore, by the first case, we get that all honest eventually output shared.

□

Lemma B.3. *Suppose an honest party P_i sends a shared message containing C_i and a distinct honest party P_j sends a shared message containing C_j . Then $C_i = C_j$.*

Proof. We prove the lemma by contradiction. Suppose $C_i \neq C_j$. P_i outputs the shared for C_i only if it has received at least $n - f$ ready messages for C_i or verified $Sig_{\mathbb{C}}$ that contains $n - f$ signed ready messages for C_i. P_j outputs the shared for C_j only if it has received at least $n - f$ ready message for C_j or verified $Sig_{\mathbb{C}}$ that contains the on $n - f$ signed ready messages for C_j. From the $n - f$ ready messages for C_i at least $n - 2f$ are generated by honest parties. From the $n - f$ ready messages for C_j at least $n - 2f$ are generated by honest parties. Since there are at most f malicious parties and $n > 3f$ this is only possible if an honest party signed two contradicting ready messages. A contradiction to the code of the protocol.

□

Lemma B.4. *The protocol in Algorithms 1 and 2 satisfy H(iii) (Correctness).*

Proof. Let J be the index set of the k honest parties that have completed the sharing and let s_j be the shares of J. To prove the first part, suppose the dealer has shared s and is honest throughout the sharing stage. Towards a contradiction assume $z \neq s$.

Because the dealer is honest, it is easy to see that every echo message sent from an honest P_i to P_j contains $C, u(i, j), u(j, i)$, as computed by the dealer. Furthermore, if the parties in J computed their shares only from these echo messages, then $s_j = a_j(0) = u(j, 0)$. But since $z \neq s$, at least one honest party P_i computed a polynomial $a_i(y) \neq u(i, y)$; this must be because P_i accepted an echo or ready message from some corrupted P_m containing $a_m \neq u(m, i)$. Since P_i has evaluated verify-point to true, we have $g^a = \prod_{j=0}^{f} (\mathbb{C}_{jl})^{i^j}$ On the other hand, the dealer has sent polynomials a_m to P_m satisfying $g^{a_m} = \prod_{j=0}^{f} (\bar{\mathbb{C}}_{jl})^{i^j}$. However, from Lemma B.3 and the fact that the dealer is honest we know that $\bar{\mathbb{C}} = \mathbb{C}$. Hence P_m knows an $a_m \neq a$ such that $g^a = g^{a_m}$, however this is a collision to the commitment scheme which should be hard (binding commitment). A contradiction.

To prove the second part, assume by a way of contradiction that two distinct honest parties P_i and P_j reconstruct values z_i and z_j such that $z_i \neq z_j$. This means that they have received two distinct sets $S_i = (l, s_l^{(i)})$ $S_j = (l, s_l^{(j)})$) and of k shares each, which are valid with respect to the unique commitment matrix $\mathbb{C}$ used by P_i and P_j (the uniqueness of $\mathbb{C}$ follows from Lemma B.3). According to the protocol, z_i and z_j are interpolated from the sets S_i, S_j respectively. Since the shares in are valid, it is easy to see that $g^{z_i} = \mathbb{C}_{00} = g^{z_j}$, however the commitment scheme is binding. A contradiction.

□

Lemma B.5. *The protocol in Algorithms 1 and 2 satisfy H(iV) (Privacy).*

Proof. If the dealer p_d is honest, it follows directly by inspection of the protocol that the dealer generated a polynomial with degree (t, f) then no set of f shares can reconstruct it. Furthermore, by inspection of the code, no honest party reveals its shares and polynomials to any unauthorized party. Finally, the lemma follows from the hiding property of the commitment scheme. That is, the adversary is unable to recover a share or points on the polynomial by looking at $\mathbb{C}$.

□

C EPCC PROOFS

Lemma 5.1. *If a valid coin for some sq is generated, then at least $2f + 1$ valid share-coins associated with some bit vector V for sq were previously generated, $f + 1$ of which by honest parties.*

Proof. By the code, P_i either gets $2f + 1$ share-coin messages with correct shares associated with some bit vector V and sq or gets a coin message with valid coin associated with some bit vector V and sq. In the second case, by the generate-coin and verify-coin functions, we know that at least $2f + 1$ valid share-coins associated with V and sq are needed to produce the valid coin. In addition, note that by the code, P_i ignores bit vectors that include less than $2f + 1$ ones. Therefore, we only need to show that the adversary cannot produce more than f valid share-coins associated with sq and some bit vector V that includes at least $2f + 1$ ones (before honest parties do it).

By the H(iv) property of HAVSS (*privacy*), the adversary cannot learn the shares of honest parties that were delivered in HAVSS instances with honest dealers. Since V includes at least $2f + 1$ ones, we get that the associated keys of honest parties include shares from HAVSS instances with honest dealers. Therefore, the adversary cannot learn the keys of honest parties that are associated with V, and thus cannot produce more than f valid share-coins associated with V.

□

Lemma 5.2. *The protocol in Algorithm 4 satisfies E(i) (Unpredictability).*

Proof. First, due to W(i) (Inclusion), we know that any valid shared private-key has contribution of at least $f + 1$ honest parties who never reveal them, hence the adversary does not know the shared private-key.

Second, consider an honest party P_i who's coin-toss(sq) invocation returns a coin ρ. By Lemma 5.1, at least $f + 1$ share-coins for sq were previously generated. By the code, an honest party does not generate a share-coin for sq before coin-toss(sq) is invoked. Therefore, the adversary can neither know the private-key nor predict ρ before at least one honest party invokes coin-toss(sq).

□

Lemma 5.3. *For every sq, if an invocation of* coin-toss(sq) *by an honest party P_i returns, then all* coin-toss(sq) *invocations by honest parties eventually return.*

Proof. Assume by a way of contradiction that some invocation of coin-toss(sq) by an honest party P_j never returns. By the code, before P_i returns, it forwards the coin to all other parties in a coin message, and thus all other honest parties eventually get this messages. In addition, since P_i is honest, we know that the coin is valid and associated with a bit vector that includes at least $2f + 1$ ones. Therefore, P_j will eventually get this coin, successfully verify it and return it. A contradiction.

□

Lemma 5.4. *The protocol in Algorithm 4 satisfies E(ii) (Termination).*

Proof. Assume by a way of contradiction that some invocation of coin-toss(sq) by an honest party P_j never returns. By Lemma 5.3, we get that no invocation of coin-toss(sq) by an honest party returns. By the W(iii)(Eventual Agreement) property of the wDKG sub-protocol, every party P_i eventually outputs an ultimate prediction and never outputs a prediction again. Moreover, by W(iii), we also know that all the ultimate predictions of honest parties are matching, meaning that they are associated with the same bit vector V'. In addition, by property W(i), we get that V' includes at least $2f + 1$ ones.

Therefore, by the code, all honest parties eventually generate and send to all other parties a valid coin-share for sq that is associated with V'. Hence, P_j will eventually get $2f + 1$ valid coin shares for sq that are associated with a valid bit vector (includes $2f + 1$ ones), and thus eventually generate a coin and return. A contradiction.

□

Lemma 5.5. *If an honest party generates a share-coin associated with V, then it will never generate a share-coin associated with $V' \not\supseteq V$.*

Proof. By the code, at any point during the EPCC algorithm, an honest party generates share-coins that are associated with the bit vector that were produced (via get-key) from the last prediction it received from the wDKG sub-protocol. By property W(ii) (Containment) of the wDKG sub-protocol, we know that predictions outputted from the wDKG are related by containment, and thus the lemma follows.

□

Lemma 5.6. *If for some sq, two* coin-toss(sq) *invocations by two honest parties return different valid coins $\rho_1 \neq \rho_2$, then there are two bit vectors V_1, V_2 s.t. (1) $V_1 \subset V_2$; and (2) $f + 1$ honest parties generated valid share-coins associated with V_1 for sq and $f + 1$ honest parties generated valid share-coins associated with V_2 for sq.*

Proof. By Lemma 5.1, ρ_1 implies that at least $2f + 1$ valid share-coins associated with some bit vector V_1 for sq were previously generated, $f + 1$ of which by honest parties; and ρ_2 implies that at least $2f + 1$ valid share-coins associated with some bit vector V_2 for sq were previously generated, $f + 1$ of which by honest parties. Therefore, there is at least 1 honest party P_i that generated a share-coin for sq that is associated with V_1 and another share-coin for sq that is associated with V_2. Since $\rho_1 \neq \rho_2$, we get by H(iii) (HAVSS correctness) that $V_1 \neq V_2$. Therefore, by Lemma 5.5, V_1 and V_2 are related by containment.

□

Lemma 5.7. *For every $1 \leq k \leq f + 1$, if there are k sequence numbers sq for which two invocations of* coin-toss(sq) *by honest parties output different coins, then there is a bit vector V of size at least $2f + 1 + k$ such that $f + 1$ honest parties generated valid share-coins associated with V.*

Proof. We prove by induction on k.

Base: we show that if there is one sq for which two invocations of coin-toss(sq) by honest parties output different coins then there is some vector V of size at least $2f + 2$ such that $f + 1$ honest parties generated valid share-coins associated with V. By the code, honest parties only generate share-coins that are associated with bit vectors that were produced from wDKG prediction outputs. Thus, by property W(i) (Inclusion) of wDKG, honest parties only generate share-coins that are associated with bit vectors of size at lest $2f + 1$. Therefore, the base case follows from Lemma 5.6.

Step: Assume the lemma holds for some $1 \leq k \leq f$, we show that the lemma holds for $k + 1$. First note that sine $k \leq f$, we get that the total number number of crytpographic signatures is polynomial in the security parameter, and thus all previous lemmas hold except with negligible probability. Let sq^k and sk^{k+1} be the k^{th} and $(k + 1)^{th}$ sequence numbers for which two invocations of coin-toss by honest parties output different coins, respectively. By the well-formed nature of EPCC we are guaranteed that any honest party invokes coin-toss(sq^{k+1}) only after coin-toss(sq^k) returns. By the induction assumption, there is a bit vector V^k of size at least $2f + 1 + k$ such that $f + 1$ honest parties generated valid share-coins associated with V^k before their coin-toss(sq^k) invocation returns. So by Lemma 5.5 and by well-formance, there are $f + 1$ honest parties that do generate share-coins associated with bit vectors with less than $2f + 1 + k$ entries

for sk^{k+1}. By Lemma 5.1, we need $2f + 1$ valid shares in order to generate a valid coin for sk^{k+1}. Thus, since every bad party can generate at most one valid share-coin, we get that only coins that are associated with bit vectors of size at least $2f + 1 + k$ can be generated for sk^{k+1}. Therefore, the lemma follows from lemma 5.6.

□

Lemma 5.8. *The protocol in Algorithm 4 satisfies E(iii) (Eventual Agreement).*

Proof. Assume by a way of contradiction that there are $f + 1$ sequence numbers sq for which two invocations of coin-toss(sq) by honest parties return different coins. By Lemma 5.7, then there is a bit vector V of size at least $3f + 2$ such that $f + 1$ honest parties generated valid share-coins associated with V. Since the number of parties is (and thus HAVSS) instances is $3f + 1$, we get a contradiction to the bit vector definition.

□

D ADKG PROOFS

Lemma D.1. *All honest parties decide* 1 *in at least $n - f$ ABA instances.*

Proof. Consider two case:

- First, there is an honest party that inputs 0 in some ABA instance. By the code, it decides 1 in at least $n - f$ ABA instances. Therefore, by the ABA Agreement property all honest parties decide 1 in at least $n - f$ ABA instances.
- Second, no honest party invoke an ABA with 0. By the H(i) (Liveness) property of HAVSS, there are $n - f$ HAVSS instance for which all honest parties deliver a share, and thus input 1 in the corresponding ABA instances. Therefore, by the Validity and Termination properties all honest parties decide 1 in these $n - f$ ABA instances.

□

Lemma D.2. *The protocol in Algorithm 5 satisfies A(i) (Validity).*

Proof. Consider a party p_i that outputs a set of shares v. By the code, since p_i outputs a value, it outputs a decision in all ABA instances. Moreover, v includes all the shares of HAVSS for which the corresponding ABA decides 1. Thus, we need to prove that p_i decides 1 in at least $n - f$ ABA instances. The Lemma follows from Lemma D.1.

□

Lemma D.3. *The protocol in Algorithm 5 satisfies A(ii) (Agreement).*

Proof. By the code, parties include all the shares of HAVSS instances for which they output 1 in the corresponding ABA instances. Therefore, the lemma follows from the ABA Agreement property.

□

Lemma D.4. *If ABA.j outputs 1, then all honest parties eventually deliver a share for the HAVSS instance for which p_j is the dealer.*

Proof. By the ABA validity, at least one honest party input 1 to *ABA.j*. Therefore there is at least one honest party who delivers a share for the HAVSS instance for which p_j is the dealer. The Lemma follows from the Agreement (Hii) property of HAVSS.

□

Lemma D.5. *The protocol in Algorithm 5 satisfies A(iii) (Liveness).*

Proof. By Lemma D.4, all parties output provided they decide in all the ABA instances. Thus, we need to prove that all ABA instance eventually terminate. Therefore, by the termination property of ABA, we only need to prove that all honest party invoke all ABA instances. Thus, by the code, we need to prove that at least $n - f$ ABA instance decide 1. The Lemma follows from Lemma D.1.

□

Lemma D.6. *The protocol in Algorithm 5 satisfies A(iv) (Privacy).*

Proof. Consider an honest party p_i that outputs a set of shares v_i. By the Validity property, $|v_i| \geq n - f$, and thus *source*(v_i) contains at least 1 honest party. The lemma follows from H(iV) (Privacy) of HAVSS.

□