Bootstrapping Consensus Without Trusted Setup: Fully Asynchronous Distributed Key Generation

Eleftherios Kokoris-Kogias[1]*, Alexander Spiegelman[2], Dahlia Malkhi[3], and Ittai Abraham[2]

[1] EPFL
eleftherios.kokoriskogias@epfl.ch
[2] VMware Research
(spiegelmans|iabraham)@vmware.com
[3] Calibra
dahliamalkhi@gmail.com

Abstract. In this paper, we present the first fully asynchronous distributed key generation (ADKG) algorithm as well as the first distributed key generation algorithm that can create keys with a dual $(f, 2f+1)-$threshold that are necessary for scalable consensus (which so far needs a trusted dealer assumption).

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. [8] on how to create an AVSS protocol with recovery thresholds $f+1 < k \leq 2f+1$, which is of independent interest. Our High-threshold-AVSS (*HAVSS*) uses an asymmetric bi-variate polynomial, where the secret shared is hidden from any set of k nodes but an honest node that did not participate in the sharing phase can still recover his share with only $n-2f$ shares, hence be able to contribute in the secret reconstruction.

Another building block for ADKG is a novel *Eventually Perfect* Common Coin (EPCC) abstraction and protocol that enables the participants to create a common coin that might fail to agree at most $f+1$ times (even if invoked a polynomial number of times). Using *EPCC* we implement an Eventually Efficient Asynchronous Binary Agreement (EEABA) in which each instance takes $O(n^2)$ bits and $O(1)$ rounds in expectation, except for at most $f+1$ instances which may take $O(n^4)$ bits and $O(n)$ rounds in total.

Using EEABA we construct the first fully Asynchronous Distributed Key Generation (ADKG) which has the same overhead and expected runtime as the best partially-synchronous DKG ($O(n^4)$ words, $O(n)$ rounds). As a corollary of our ADKG we can also create the first Validated Asynchronous Byzantine Agreement (VABA) in the authenticated setting 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 $O(n)$ time bootstrap via ADKG.

1 Introduction

Byzantine agreement is one of the fundamental problems in distributed fault-tolerant computing. It consists of n communicating parties, at most f of which are corrupted, who want to agree on a common valid value. If the set of values is $\{0, 1\}$, then we call it Binary Agreement, whereas if we want to guarantee a decision on some partys input, provided it satisfies a globally verifiable external validity condition, we call it Validated Agreement. More specifically our core focus is on Distributed Key Generation (DKG) [26] a protocol that requires Byzantine Agreement to work. A DKG is a protocol that builds on top of secret sharing [13] to generate a threshold-protected private-public key-pair. It works in two steps, first, each party secret shares a locally generated private-public key-pair and then the parties agree on a subset of at least $f+1$ dealers whose secret-sharing invocation terminated correctly. A shared key that is generated by a DKG can later be used as a threshold key to encrypt data under the control of the parties [28,21], to generate constant sized signatures representing a threshold of parties (used in efficient BFT protocols [29,17]), to scale blockchains [22,30,2], or to generate strong random coins (used in fully asynchronous (V)ABA protocols [10,8,1]).

* Work done at VMware Research

Currently, the only DKG that does not assume synchrony is Hybrid-DKG [19]. Yet, Hybrid-DKG assumes weak synchrony and can only generate keys with a reconstruction threshold of $k = f + 1$. As a result it can neither be used to bootstrap efficient asynchronous (V)ABA [24,10,1,8] (because of the network assumptions) nor efficient partially synchronous consensus (because of the low reconstruction threshold) that is currently being deployed in the wild, such as the Hotstuff [29] or the SBFT [17] variant deployed by VMWare [16]. In this paper, we solve this problem of trustless bootstrapping.

Formally, the main theorem we prove in this paper is:

Theorem 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 $O(n^4)$ word communication and expected $O(n)$ running time.*

Building blocks

Our first building block towards ADKG is an efficient, Asynchronous Binary Agreement protocol that does not assume a trusted setup (Section 6.1). We call it *Eventually Efficient* ABA as it might have $f + 1$ failed runs before converging, but once it converges it is optimal. Formally EEABA achieves the following Lemma:

Lemma 1. *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 $O(n)$ one-shot ($O(1)$ amortized) running time.*

Our second building block (Section 3) is called *High-threshold Asynchronous Verifiable Secret Sharing* (HAVSS). It is an extension of Cachin et al. [8] 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 allow to increase to k) from the recovery threshold (which we keep at $f + 1$). In order to encode this change, we use an *asymmetric* bivariate polynomial where each dimension plays a different role (recovery, reconstruction) and to defend against an adaptive adversary we add a reliable broadcast step before terminating the sharing successfully. More formally HAVSS satisfies the following lemma.

Lemma 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": A third (intermediate) building block is called *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 deterministic 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 in the common case. 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 during a wDKG invocation.

Our final building block (Section 5) is called an *eventually perfect common coin* (EPCC). It relies on the wDKG to detect the points of agreement and on adaptively secure deterministic threshold signatures [23] to produce the randomness. The key property of the EPCC is that the adversary can only force it to disagree a finite ($f + 1$) 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 date 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 $f + 1$ 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. [24] to create EEABA (Lemma 1). Afterwards, we can invoke it n times (one for every HAVSS) to implement ADKG (Theorem 1) and terminate the bootstrap phase of VABA (Corollary 2).

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. *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 $O(n)$ running time.*

as well as, through the composition of our ADKG, that bootstraps the threshold signature and the randomness generation scheme, with the VABA of Abraham et al [1]:

Corollary 2. *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 [8] affirmatively. Although we use it in a fully asynchronous environment HAVSS can also remove the trusted setup requirement of Hotstuff and SBFT, by combining HAVSS with Hybrid-DKG [19].
- Using HAVSS we introduce a novel *eventually perfect* common coin construction that disagrees at most $f + 1$ times but can be used polynomial times.
- Using our EPCC inside the protocol of Moustefauoi et al. we create EEABA protocol that needs no trusted setup. It terminates in($O(n)$) 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 an efficient, leaderless, fully asynchronous DKG. 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 main building blocks of VABA. The ADKG has $O(n^4)$ word complexity and terminates in an expected $O(n)$ rounds.

2 Definitions and Model

In order to reason about distributed algorithms in cryptographic settings we adopt the model defined in [9,10,1]. 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 in an execution is called honest. To be able to use the threshold signature from [23] we assume the cryptographic random oracle model and we treat hash functions as random oracles.

Computation. Following [10,9], 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. 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 at some prior time. This assumption is reasonable since it can be easily implemented with standard cryptographic techniques in our model.

Termination. Note that the traditional definition of the liveness property in a 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. [10,9], 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 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 [10,9]:

Definition 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. [10,9] 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. [10,9] require another property:

– If all messages sent by honest parties have been delivered, then all honest parties terminated.

In this paper, we consider both efficiency and termination properties as defined in [10,9].

Complexity. We use the following standard complexity notions (see for example Cannetti and Rabin [11]). We measure the expected word communication of our protocol as the maximum over all inputs and applicable adversaries of the expected total number of words sent by honest parties where expectation is taken over the random inputs of the players and of the adversary. We assume a finite domain V of valid values for the Byzantine agreement problem and say that a word can contain a constant number of signatures. We measure the expected running time of our protocol as the maximum over all inputs and applicable adversaries of the expected duration where expectation is taken over the random inputs of the players and of the adversary. The duration of an execution is the total time until all honest players have terminated divided by the longest delay of a message in this execution. Essentially the duration of an execution is the number of steps taken if this execution is re-run in lock-step model where each message takes exactly one time-step.

Following Cachin et al. [9] (see Lemma 1 therein), in order to show that our view-based protocol runs in an expected constant running time and has expected $O(n^4)$ word communication, it is enough to show that:

– every view consists of $R(k) = O(n^3)$ messages that consist of $O(n)$ words, and
– the total number of messages is probabilistic uniformly bounded by R.

Cryptographic Abstractions Given that our protocols use cryptographic constructions as black boxes we 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. [10] This way the reader can focus on the distributed protocol which is the novelty. Nevertheless, the actual implementation of our consensus algorithm requires pairing-based threshold cryptography as shown by Libert et al. [23] in order to be adaptively-secure.

Diffie-Hellman Based Coin In this section, we briefly present the coin-tossing protocol of Cachin et al. [10] for completeness. 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. It uses this share to generate a share of the coin $F(C)$ which is $\bar{g}^{x_i}$, along with a "validity proof"[4].

For our purpose we abstract the inner workings of the coin and only expose 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 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

AVSS [8,11,5] schemes provide a recovery 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 $n - f - f = n - 2f$.

To guarantee agreement about completing the sharing, nodes can reliably broadcast ready messages. Here we adopt an alternative approach (which makes our protocol simpler), where nodes sign their ready messages, and when a node collects $n - f$ signed ready messages, it broadcasts the set as a proof of completion.

3.1 Definition

Our protocol falls in the class of *dual-threshold sharing* [10], which are protocols that allow the reconstruction threshold of a secret to be more than $f + 1$. Although in the original AVSS [8] 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[5] efficient Byzantine agreement protocols [1,29] which currently require a trusted dealer during setup.

We follow the definitions of Cachin et al [8] 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.

[4] Discrete log equality NIZK-proof [12]
[5] Coupled with a suitable DKG [19]

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 $(ID.d, \mathsf{in}, \mathsf{share}, s)$. 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 $(ID.d, \mathsf{out}, \mathsf{shared})$. 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. 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 $(ID.d, \mathsf{in}, \mathsf{reconstruct})$. 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 $(ID.d, \mathsf{out}, \mathsf{reconstructed}, z_i)$. 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 $f+1$ honest parties have completed the sharing of $ID.d$, there exist a fixed value z such that the following holds:

1. If the dealer has shared $(ID.d, \mathsf{in}, \mathsf{share}, s)$ 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 $(ID.d, \mathsf{in}, \mathsf{share}, s)$ 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 of HAVSS

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 $\mathbb{G}$.

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.

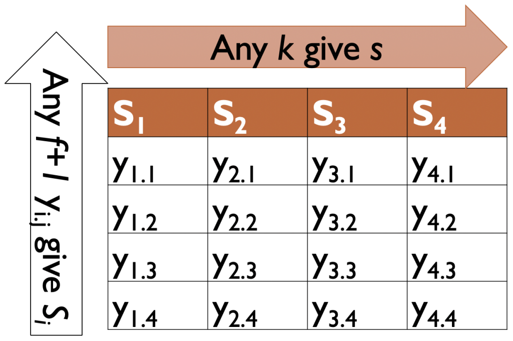
Any k give s
Any f+1 y_{i.j} give S_i
S_1 S_2 S_3 S_4
y_{1.1} y_{2.1} y_{3.1} y_{4.1}
y_{1.2} y_{2.2} y_{3.2} y_{4.2}
y_{1.3} y_{2.3} y_{3.3} y_{4.3}
y_{1.4} y_{2.4} y_{3.4} y_{4.4}

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 delear 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 $s = u(0,0)$.

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 $s = u(0,0)$. A detailed description of the protocol is given in 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 that the set of signatures Sig_C area valid.

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.

The protocol described above has communication complexity of $O(n^4)$, however, it can be optimized to $O(n^3)$ as shown in [8].

3.3 Analysis

Proofs for the properties defined earlier follow from [8] with the below changes for agreement to hold in HAVSS.

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.

This completes the proof of Lemma 2.

3.4 Discussion

Other variants of HAVSS: Although we described a specific version of HAVSS by modifying AVSS [8], the core idea can be used for other AVSS protocols. For examples we can decrease the common-case overhead to $O(n^2)$ by using the techniques of Basu et al. [5], where they only create 4 recovery polynomials instead of n and we could decrease another $O(n)$ by applying the novel commitment scheme of Kate et al. [20], if we are willing to accept a trusted setup assumption. Each recover polynomial would have $n-2f$ reconstruction threshold, while the actual secret could still have $n-f$.

These approaches are interesting but do not help our main goal of ADKG, since we care about the worse case communication complexity (which is the same for AVSS and sAVSS) and we cannot have a trusted setup assumption as this is the assumption ADKG manages to lift.

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

```
1:  upon initialization do
2:      $\mathsf{success} \leftarrow \mathsf{false}$
3:      for all $\mathbb{C}$ do
4:          $e_C \leftarrow 0; r_C \leftarrow 0; f_C \leftarrow 0$
5:          $A_C \leftarrow \emptyset;\ B_C \leftarrow \emptyset\ Sig_C \leftarrow \emptyset$

6:  upon receiving "$ID.d, \mathsf{in}, \mathsf{share}, s$" do                              ▷ only $P_d$
7:      choose a random asymmetric bivariate polynomials $u$ of degree (t, f) with $u(0,0) = u_{00} = s$, i.e.,
            $$u(x,y) = \sum_{j,l=0}^{t,f} u_{jl} x^j y^l$$
8:      $\mathbb{C} \leftarrow \{C_{jl}\}$, where $C_{jl} = g^{u_{jl}}$ for $j \in [0,t]$ and $l \in [0,f]$
9:      for $j \in [1,n]$ do
10:         $a_j(y) \leftarrow u(j,y);\ b_j(x) \leftarrow u(x,j)$
11:         send "$ID.d, \mathsf{send}, \mathbb{C}, a_j, b_j$" to $P_j$

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

15: upon receiving "$ID.d, \mathsf{echo}, \mathbb{C}, \alpha, \beta$" from $P_m$ for the first time do
16:     if $\mathsf{verify-point}(\mathbb{C}, i, m, \alpha, \beta)$ then
17:         $A_C \leftarrow A_C \bigcup \{(m,\alpha)\}; B_C \leftarrow B_C \bigcup \{(m,\beta)\}$
18:         $e_C \leftarrow e_C + 1$
19:         if $e_C = k$ then
20:             interpolate $\bar{a}, \bar{b}$ from $B_C, A_C$, respectively
21:             for $j \in [1,n]$ do send "$ID.d, \mathsf{ready}, \mathbb{C}, \bar{a}(j), \bar{b}(j), sig_i$" to $P_j$

22: upon receiving "$ID.d, \mathsf{ready}, \mathbb{C}, \alpha, \beta, sig_m$" from $P_m$ for the first time do
23:     if $\mathsf{verify-point}(\mathbb{C}, i, m, \alpha, \beta)$ then
24:         $Sig_C \leftarrow Sig_C \bigcup \{(m, sig_m)\}$
25:         $r_C \leftarrow r_C + 1$
26:         if $r_C = n - f$ and $e_C \geq k$ then
27:             $\bar{\mathbb{C}} \leftarrow \mathbb{C};\ s_i \leftarrow \bar{a}(0);\ \mathsf{success} \leftarrow \mathsf{true}$
28:             for $j \in [1,n]$ do send "$ID.d, \mathsf{shared}, \mathbb{C}, Sig_C, \bar{b}(j)$" to $P_j$
29:             output $(ID.d, \mathsf{out}, \mathsf{shared})$

30: upon receiving "$ID.d, \mathsf{shared}, \mathbb{C}, Sig_C^m, \beta$" from $P_m$ for the first time do
31:     if $\mathsf{verify-shared}(\mathbb{C}, Sig_C^m)$ then
32:         if $e_C \geq k$ then                                                    ▷Can fully terminate
33:             $\bar{\mathbb{C}} \leftarrow \mathbb{C};\ s_i \leftarrow \bar{a}(0);\ \mathsf{success} \leftarrow \mathsf{true}$
34:             for $j \in [1,n]$ do send "$ID.d, \mathsf{shared}, \mathbb{C}, Sig_C, b(\bar{j})$" to $P_j$
35:             output $(ID.d, \mathsf{out}, \mathsf{shared})$
36:         else if $\mathsf{verify-point}(\mathbb{C}, i, m, \beta)$ then                    ▷Can only recover share
37:             $B_C \leftarrow B_C \bigcup \{(m,\beta)\}$
38:             $r_C \leftarrow r_C + 1$
39:             if $r_C = f + 1$ then
40:                 $\bar{\mathbb{C}} \leftarrow \mathbb{C}$
41:                 interpolate $\bar{a}$ from $B_C$,
42:                 $s_i \leftarrow \bar{a}(0)$
43:                 output $(ID.d, \mathsf{out}, \mathsf{shared})$
```

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, $\sigma$" from $P_m$ do
5:     if verify − share($\bar{\mathbb{C}}, m, \sigma$) 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
```

HAVSS for Bootstrap of Hotstuff/SBFT Although this paper focuses on fully asynchronous protocols, advancements in partially synchronous protocols [29,17] 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 [19] 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 [14] 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.

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 predictions 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 $\mathit{shares}_i(S) \triangleq \{s_i^j \mid P_j \in S\}$, and given a prediction $\mathbb{P}$ of P_i, $\mathit{source}(\mathbb{P}) \triangleq \{P_j \mid s_i^j \in \mathbb{P}\}$. Note that $\mathit{source}(\mathit{shares}_i(S)) = S$. We say that two predictions $\mathbb{P}_1, \mathbb{P}_2$ of different parties are *matching* if $\mathit{source}(\mathbb{P}_1) = \mathit{source}(\mathbb{P}_2)$.

The wDKG protocol provides the following properties.

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

W(ii): Containment. For each party P_i, predictions are ordered by strict containment, $\forall k < j : \mathbb{P}_k \subset \mathbb{P}_j$.

W(iii): Eventual Agreement. Every honest parties 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 cannot 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. 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 waits until $H \supseteq shares_i(S)$ and then outputs the prediction $shares_i(S)$ provided it did not output a prediction $\mathbb{P} \not\subset shares_i(S)$ before. Since parties output increasing predictions by containment and since the smallest output prediction consists of at least $2f + 1$ shares, we get that each party outputs at most $f + 1$ predictions.

Note that an honest party might broadcast up to $f + 1$ candidate-key messages, but a Byzantine party might broadcast an exponential number of such messages. Even for honest parties, there may be a quadratic number of candidate messages. Therefore, in order to avoid an exponential (or even a quadratic) number of predicitions (1 for every source set we get) 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 $j \in \{1, \dots, n\}$ do
3:         $S_j \leftarrow \{\}$                          ▷The source (set of parties) $p_i$ received from $p_j$
4:     $H \leftarrow \{\}$                                ▷ The set of HAVSS shares $p_i$ outputs
5:     $S_{\mathbb{P}} \leftarrow \{\}$                   ▷The source set of the current prediction
6:     $C[:] \leftarrow 0$                                ▷A counter for every possible source
7:     select random $r_i$
8:     invoke $(i, \mathsf{in}, \mathsf{share}, r_i)$     ▷Every party starts an HAVSS as a dealer

9: upon $(ID.j, \mathsf{out}, \mathsf{shared})$ do
10:    $H \leftarrow H \cup \{s_i^j\}$
11:    if $|H| \geq n - f$ 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 $S \supset S_j \cup S_{\mathbb{P}}$ then
15:        $S_j \leftarrow S$
16:        $C[S] \leftarrow C[S] + 1$
17:        if $C[S] = n - f$ then
18:            $S_{\mathbb{P}} \leftarrow S$
19:            wait until $H \supseteq shares_i(S)$
20:            output $(\mathsf{out}, \mathsf{key}, shares_i(S))$
```

4.3 Analysis

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

Lemma 3. *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. *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, $|source(\mathbb{P})| = |\mathbb{P}| \geq n - f$.

Lemma 5. *An honest party 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 $H \supseteq shares_i(S)$ 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 $H \supseteq shares_i(S)$, and thus P_i will eventually end the waiting in Line 19.

Lemma 6. *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 lest $n - f$ 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 5 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 $shares_i(S_H)$.

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 7. *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. Furthermore, we assume that is a party invoke 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 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 check 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). Therefore, 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 in Algorithm 4, in which we use the functions below.

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:

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

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 $K^i_{\mathbb{P}}$ 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 [10], which yields a common coin flip for each sq input. In our educational example, we use Pedersen [26] DKG, which does not produce uniformly random keys [15], but as shown by Libert et al. [23] 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 schemes provide, and more details and formal proofs can be found in [10]. 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 scheme.

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 associated 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.

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.

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

7: upon $(\mathsf{out}, \mathsf{key}, \mathbb{P})$ do                    ▷prediction output form the wDKG sub-protocol
8:     $\langle K, V\rangle \leftarrow \mathsf{get\text{-}key}(\mathbb{P})$
9:     if $currentSQ \neq \bot$ then
10:        BroadCastShare()

11: upon coin-toss(sq) do
12:     $currentSQ \leftarrow sq$                    ▷Avoid races during concurrent invocations
13:     if $\exists V'$ s.t. $Coins[\langle V', sq\rangle] \neq \bot$ then
14:         ForwardCoinAndReturn$(V', sq)$
15:     if $\exists V'$ s.t. $|S[\langle V', sq\rangle]| \geq 2f+1$ then
16:         BroadCastCoinAndReturn$(V', sq)$
17:     if $V \neq \bot$ then
18:         BroadCastShare()

19: upon receiving "$\mathsf{share}, sq, \sigma, V_j$" message from party $P_j$ for the first time do
20:     $C \leftarrow \mathsf{get\text{-}commitment}(V_j)$
21:     if $\mathsf{verify\text{-}share}(C, sq, j, \sigma) \ \wedge \ \sum_{k=1}^{n} V_j[k] \geq 2f+1$ then
22:         $S[\langle V_j, sq\rangle] \leftarrow S[\langle V_j, sq\rangle] \cup \{\sigma\}$
23:         if $sq = currentSQ \ \wedge \ |S[\langle V_j, sq\rangle]| \geq 2f+1$ then
24:             BroadCastCoinAndReturn$(V_j, sq)$

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

31: procedure BroadCastShare()
32:     $C \leftarrow \mathsf{get\text{-}commitment}(V)$
33:     $\sigma \leftarrow \mathsf{generate\text{-}share}(C, K, currentSQ)$
34:     send "$\mathsf{coin\text{-}share}, currentSQ, \sigma, V$" to all parties

35: procedure BroadCastCoinAndReturn$(V', sq)$
36:     $C \leftarrow \mathsf{get\text{-}commitment}(V')$
37:     $\rho \leftarrow \mathsf{generate\text{-}coin}(C, S[\langle V', sq\rangle], sq)$
38:     send "$\mathsf{coin}, sq, \rho, V'$" to all parties
39:     $currentSQ \leftarrow \bot$
40:     output $(\mathsf{out}, \mathsf{coin}, sq, \rho)$

41: procedure ForwardCoinAndReturn$(V', sq)$
42:     send "$\mathsf{coin}, sq, Coins[\langle V', sq\rangle], V'$" to all parties
43:     $currentSQ \leftarrow \bot$
44:     output $(\mathsf{out}, \mathsf{coin}, sq, Coins[\langle V', sq\rangle])$
```

5.3 Analysis

Correctness proof In this section we show *unpredictability, termination, and eventual agreement* of our EPCC.

Lemma 8. *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 valid 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 9. *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 8, 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 10. *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 11. *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 10, 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 12. *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 13. *If for some sq, two* coin-toss(sq) *innovations 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 8, 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 that $V_1 \neq V_2$. Therefore, by Lemma 12, V_1 and V_2 are related by containment.

Lemma 14. *For every* $1 \leq k$*, 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 than 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 form 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 13.

DM: no, this doesn't prove the base case :-) because it doesn't use the sequence number seq at all. This shows that two coin-tosses for ANY sequence number differ in their bit vector, not that for $k=0$ there are $2f+k+2$ bits in the larger bit vector. This is the error. **sasha:** I do not understand the problem. The base case is for $k=1$, and lemma 13 talks about any sq. I think the base case is fine, but we do need to use well-formance in the step proof (as we now do)

Step: Assume the lemma holds for some $1 \leq k$, we show that the lemma holds for $k+1$. 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 12 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 8, 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 13.

Lemma 15. *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 14, 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.

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 [24,7]. The most efficient asynchronous BA solution is from Moustefaoui's et al [24] and has O(n^2) bit complexity when the coin is perfect [6].

Since our coin has at most f bad flips, when we plug it in [24], then 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 $O(n)$ rounds. We refer to an ABA that has this succession property as eventually efficient ABA (EEABA).

We refrain from re-introducing 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 [24] which (as mentioned above) has an overall complexity remains $O(n^3)$ bit complexity and expected $O(n)$ rounds. Hence the total complexity of EEABA is $O(n^3)$ bit complexity and expected $O(n)$ 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.

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 using a protocol that solves the more generic *Asynchronous Common Subset (ACS)* problem introduced by Ben-or et al [6]. In an ACS protocol, n processors have some initial value and they need to agree on a 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.

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. The protocol outputs a private set of shares v that are matching (have the same *source*) for all honest parties.

The ADKG protocol provides the following properties, except with negligible probability:

A(i): Validity. If a correct party outputs a set of shares v, then $|v| \geq n-f$ and v includes only valid shares from at least $n-2f$ correct dealers.

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 no honest party has revealed its private share then the adversary cannot compute the shared secret s. This is equivalent to the HAVSS privacy property defined before.

[6] They do not give an implementation for the coin, but instead use an external oracle.

Technical Overview We follow the ACS solution of Ben-Or et al [6], 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 a worst-case running time of $O(log(n))$, but since EEABA has a worst-case running time of $O(n)$ the total running time of the protocol is $O(n + logn) = O(n)$. 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 yes.
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.

Complexity 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 initial 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 the *inclusion* and the *privacy* properties of the wDKG. With this we finish the proof our main Theorem.

7 Related

Consensus is one of the most well studied distributed systems problem, first introduced by Pease et al [25]. 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 [18], non-blocking atomic commit [3], and state machine replication [27]. Algorithms that solve consensus vary much depending on the assumptions that are made about the system. 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).

ABA: The first optimally resilient ($f < n/3$) ABA was introduced by Bracha [7]. It is based on using locally drawn random coins in order 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. Canneti 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 $O(n^8 logn)$ bits. Advancements in the information theoretic secure model have lowered the cost down to $O(n^6)$ [4].

Algorithm 5 Protocol ADKG for party P_i

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

7: upon $(ID.j, \mathsf{out}, \mathsf{shared})$ do       ▷The sharing phase of $P_j$'s HAVSS completed
8:     if $P_j \in I$ then
9:         invoke $ABA.j$ with 1
10:        $I \leftarrow I \setminus \{P_j\}$

11: upon $(ABA.j, \mathsf{deliver}, 1)$ do
12:     $K \leftarrow K \cup \{s_i^j\}$            ▷This might block until the HAVSS delivers, but it will eventually terminate.
13:     $c \leftarrow c + 1$
14:     if $c = n - f$ then
15:         for all $P_l \in I$ do
16:             invoke $ABA.l$ with 0
17:             $I \leftarrow I \setminus \{P_l\}$
18:     if $c = n$ then
19:         output $K$

20: upon $(ABA.j, \mathsf{deliver}, 0)$ do
21:     $c \leftarrow c + 1$
22:     if $c = n$ then
23:         output $K$
```

In order to reduce the communication complexity, Cachin et al [10] showed how to achieve consensus against a computationally bound adversary using cryptography. Trying to achieve this, however, introduced a new assumption of a trusted dealer that deals a perfect common-coin. Moustefaoui et al. [24] slightly weekend the assumption of Cachin et al. while maintaining the same communication complexity, 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 main contributions, we build an eventually perfect common coin without the need of a trusted dealer. Our coin falls in-between the weak coin and the perfect coin and as a result, can power Mousteafaoui's protocol.

DKG: A distributed key generation is a protocol that is run 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. Finally, these values are combined and the final result is a threshold private-public key-pair that can be used for efficient ABA [10] and VABA [1]. The first DKG was proposed by Pedersen [26] and is fully synchronous. Gennaro et al. [15] 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. Finally, Kate et al. [19] realized that synchronous protocols are not suitable for large scale deployment over the internet and propose 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 in 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 [29,17] and second, we create the first fully asynchronous DKG that also has $O(n^4)$ word complexity making it practical to generate distributed keys without the need for timing assumptions.

VABA: The VABA problem was introduced by Cachin et al. [9] 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 protocols assume a perfect-coin, hence require a trusted setup. Our contribution in this model is first that we can implement a VABA protocol with $O(n^4)$ world complexity and $O(n)$ expected termination in rounds and second that we can bootstrap the more efficient protocols with our ADKG in order to get an optimal VABA if we amortize the cost of the ADKG over $O(n^2)$ runs.

8 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.

References

1. Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Validated asynchronous byzantine agreement with optimal resilience and asymptotically optimal time and word communication. *CoRR, abs/1811.01332*, 2018.
2. Georgia Avarikioti, Eleftherios Kokoris Kogias, and Roger Wattenhofer. Brick: Asynchronous state channels. *arXiv preprint arXiv:1905.11360*, 2019.
3. Ozalp Babaoglu and Sam Toueg. Understanding non-blocking atomic commitment. *Distributed systems*, pages 147–168, 1993.
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. Soumya Basu, Dahlia Malkhi, Mike Reiter, and Alin Tomescu. Asynchronous verifiable secret-sharing protocols on a good day. *CoRR*, abs/1807.03720, 2018.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC*, volume 93, pages 42–51. Citeseer, 1993.
12. David Chaum and Torben Pryds Pedersen. Wallet databases with observers. In *Annual International Cryptology Conference*, pages 89–105. Springer, 1992.
13. Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 427–438. IEEE, 1987.
14. Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. Technical report, Massachusetts Inst of Tech Cambridge lab for Computer Science, 1982.
15. 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.
16. Guy Golan Gueta. Meet project concord: An open source decentralized trust infrastructure, 2018.

17. Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure. *arXiv preprint arXiv:1804.01626*, 2018.
18. Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.
19. Aniket Kate, Yizhou Huang, and Ian Goldberg. Distributed key generation in the wild. *IACR Cryptology ePrint Archive*, 2012:377, 2012.
20. Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 177–194. Springer, 2010.
21. 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.
22. 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.
23. 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.
24. 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.
25. Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.
26. 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.
27. Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.
28. Victor Shoup and Rosario Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–16. Springer, 1998.
29. Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus in the Lens of Blockchain. *arXiv preprint arXiv:1803.05069*, 2018.
30. Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 931–948. ACM, 2018.