Appears in *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*, 1996

# Distributed Pseudo-Random Bit Generators— A New Way to Speed-Up Shared Coin Tossing

Mihir Bellare*  Juan A. Garay†  Tal Rabin‡

February 1996

**Abstract**

A shared coin is one which $n$ players "simultaneously" hold and can later reveal, but no sufficiently small coalition can influence or à priori predict the outcome. Such coins are expensive to produce, yet many distributed protocols (including broadcast and Byzantine agreement) need them in bulk.

We introduce a new paradigm for obtaining shared coins. We suggest *distributed, pseudo-random bit generators* (D-PRBGs). Analogous to a pseudo-random bit generator, which is an efficient algorithm to expand a short random seed into a long random looking sequence, a D-PRBG is a protocol which "expands" a "distributed seed," consisting of shared coins, into a longer "sequence" of shared coins, at low amortized cost per coin produced. Our main result is the construction of a D-PRBG in which this amortized cost (computation and communication) is significantly lower than the cost of any "from-scratch" shared coin generation protocol.

Furthermore, for applications which are executed repeatedly, we suggest *bootstrapping*: each run of the D-PRBG produces not only the coins for the current execution but also the seed for the next execution. Since the cost of the initial seed can now effectively be neglected, we get very fast coin generation.

Underlying these constructions are some techniques of independent interest. We consider batch Verifiable Secret Sharing (VSS), where we need to do a large number of VSSs simultaneously. We provide a method in which the amortized cost per VSS is significantly lower than the cost of a VSS for any known VSS protocol.

* Department of Computer Science & Engineering, Mail Code 0114, University of California at San Diego, 9500 Gilman Drive, La Jolla, CA 92093. E-mail: `mihir@cs.ucsd.edu`. URL: `//www-cse.ucsd.edu/users/mihir`.

† CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, and IBM T.J. Watson Research Center, PO Box 704, Yorktown Heights, New York 10598. E-mail: `garay@cwi.nl`, `garay@watson.ibm.com`. URLs: `//www.cwi.nl`, `//www.research.ibm.com/security`.

‡ MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA. E-mail: `talr@theory.lcs.mit.edu`. Work supported by an NSF Postdoctoral Fellowship. Part of this work was done when the author was visiting the IBM T.J. Watson Research Center.

# 1 Introduction

Powerful applications in fault-tolerant distributed computing are today being held up by the inefficiency of existing protocols. We wish to try to address this problem. We focus on a fundamental distributed primitive, namely Shared Coins. It is an enabler of many applications, including broadcast. Thus, more efficient protocols for shared coins would translate into efficiency improvements across a wide range of applications.

This primitive has received considerable attention in the literature. However, although of an elegant technical nature, most of the solutions are not amenable to practical settings, either because they incur high computation and communication costs, or because they assume strong underlying primitives (e.g., the existence of a broadcast channel, which the primitive itself is trying to help implement).

We focus on real situations. Usually, an execution of an application using shared coins, needs not one, but many coins. Furthermore, a distributed application is typically executed not once, but regularly, at intervals, as parties need it. (That's why it is called an application.) So coins are needed regularly.

We suggest and construct a new primitive which we call a *distributed pseudo-random bit generator.* Our construction enables us to "stretch" a small number of distributed (shared) coins to a large number of them at a low amortized cost per produced coin. We then show how a "boot-strapping" use of this generator is an effective way to keep alive a source of shared coins in applications which are being regularly executed, and thereby significantly lower the cost of these applications.

## 1.1 Distributed pseudo-random bit generators

DISTRIBUTED COINS. We are interested not in normal coins, but in distributed ones, otherwise known as *shared coins.* We have a distributed system of $n \geq 4$ players, some of which can be (maliciously) faulty. A shared coin protocol is one which the players can run to get a random binary output, not known to any of them beforehand. All players in the system view the same coin (this property is called *unanimity*), and no subset of players smaller than a given size would have any influence on the outcome of the coin. Typically, the coin is generated at some time, and kept secret until some later time at which the players want to reveal it. We stress that no one player (or even small set of players) knows the coin; it is held "jointly."

Shared coins are needed, amongst other things, for Byzantine agreement (BA) and broadcast. However, they are expensive to produce. Refer to Section 1.4 for history and discussion of existing implementations.

D-PRBGS. Recall that a pseudo-random bit generator (PRBG) is an efficient algorithm which takes a short random "seed" and stretches it to produce a longer, random-looking sequence. We recall that the motivation for these objects is that high quality random bits are hard to produce. Thus, we produce a small number of them (the seed) and then "expand" them via some cheap process. The use of PRBGs is now standard, both in practice and in theory, and many of them are known.

The situation with distributed coins is the same. A distributed coin is expensive to produce. If we need lots of them, it would be a lot of work to produce each one individually from scratch. It is thus natural to adopt the same paradigm as the above. Namely, we define and design a *distributed pseudo-random bit generator* (D-PRBG). This is a distributed protocol which takes a small number of shared coins (a distributed seed) and efficiently "expands" it to produce a large number of shared coins.

We stress that a D-PRBG is a distributed protocol. Its "input" is a "distributed input" consisting of some shared coins. (That is, each player has some local input and these jointly define the shared coins.) Similarly the "output" is a distributed output consisting of (a larger number of) shared coins. (Again, this means each player gets local outputs, and these local outputs together define the new shared coins.)

We stress that efficiency is crucial. The whole point of a PRBG is that stretching is more efficient than generating random bits from scratch; otherwise you don't need a PRBG. Similarly, we want that the "distributed stretching" protocol of a D-PRBG be more efficient, per coin generated, than from scratch methods of getting shared coins. This is what we achieve.

MAIN RESULT. Our main result is the construction of an efficient D-PRBG. It takes a distributed seed of length $O(k)$ (i.e., a set of $O(k)$ shared coins) and "expands" it to a sequence of bits (each "bit" being a shared coin) of length $kM$ at an amortized cost of $O(n^2 \log k)$ basic operations (e.g., addition of $k$-bit numbers) per coin and amortized communication of $O(n)$ messages, where $k$ is a security parameter. Furthermore, the coins are unanimous with probability $1 - Mn2^{-k}$. Thus, $M$ can be exponentially (in $k$) large. As we will see in Section 1.4, the efficiency here compares favorably with the best known methods for shared coin generation from scratch.

We also consider $k$-ary (rather than binary) coins. (A $k$-ary shared coin is a random variable distributed uniformly in $\{0,1\}^k$ and having the same properties as binary shared coins.) We provide a $k$-D-PRBG which takes as a seed $O(1)$ $k$-ary coins and produces as output $M$ $k$-ary coins, with the same costs and features as above.

In comparison to the usual (non-distributed) PRBGs, we note that unlike the cryptographic generators of [7, 19], we do not rely on complexity assumptions. (We do assume that the parties can locally produce random bits as needed. Perhaps they will do this using cryptographic pseudo-random generators, in which case complexity assumptions appear. But our process views the ability to produce local random bits as given; our focus is stretching a few shared random bits into many shared random bits.)

This result straightaway yields speed-ups in many applications including broadcast and Byzantine agreement. We now suggest another paradigm for exploiting D-PRBGs which in practical settings speeds up applications even more.

### 1.2 Bootstrapping

We consider an application which uses shared coins. We could use a D-PRBG in each invocation, paying a possibly high price to generate the distributed seed, but then getting lots of coins relatively quickly. This will indeed yield savings. But in many settings, we can do even better. The settings in question are ones where the application is executed repeatedly. We suggest bootstrapping. An initial distributed seed is generated via some known, not necessarily fast protocol. Then the generator is run to produce as many coins as the current execution of the application needs, plus another (distributed) seed. The new seed is stored until the next execution of the application.

Namely, we assume that initially $O(1)$ sealed $k$-ary coins are available, and then we use these coins to generate (exponentially many) more coins. The coins are generated in batches, according to need. Once the number of remaining coins drops beneath a certain level, a new batch is generated exploiting the (small amount of) remaining coins. The bootstrap approach to coin generation is shown in Fig. 1.

Our D-PRBG requires an expected constant number of executions of a Byzantine agreement protocol, in a model where broadcast is not assumed. If a randomized `BA` protocol is used, then the coins needed by the `BA` protocol must be taken into consideration when setting the level of coins

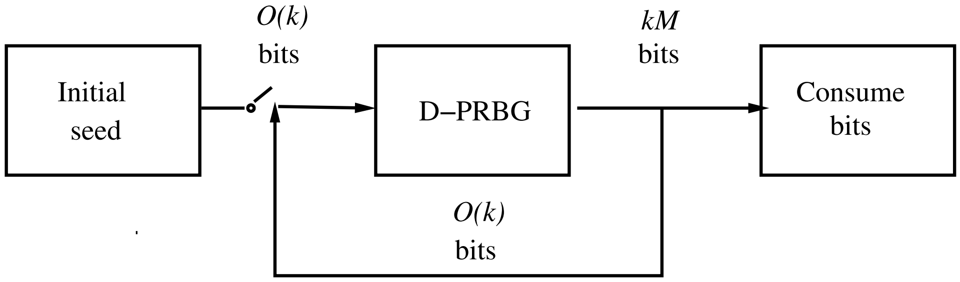


Figure 1: The bootstrap approach to distributed coin generation.

needed for the bootstrapping mechanism. For simplicity, we shall assume in this presentation that deterministic BA is carried out.

The initial set of coins can be obtained from a trusted third party, as in the case of Rabin [17], or through other pre-processing methods (for example, the interpolation of a number $m$ of polynomials, where $m$ is at least as big as the number of faulty players to be tolerated). Regarding the former, we remark that in our approach the services of a trusted dealer would be used only once, and for a small number of coins. In contrast, as the coins are "expendable," the approach of [17] requires the dealer to continuously provide them. In that sense, our method is *self-sufficient* once it gets kicked off. (In fact, we envision an adaptive mechanism, in which coins are generated on demand, with a constant threshold triggering the generation of new coins.)

Regarding the latter method, we note that the cost of the pre-processing step can be amortized over the entire execution of the system. Attempts of this nature already exist in the literature, (e.g., [1] for the specific case of deterministic agreement; [13] for the construction of a Brachia assignment; etc.). However, these amortization efforts work subject to the proviso that the set of faulty players remain (relatively) fixed. In contrast, this is not required by our method. In fact, one of the motivations and applications of our work is pro-active security (e.g., [8, 16]), which deals with settings where intruders are allowed to move over time. Our solution to multiple-coin generation can be easily adapted to this scenario.

### 1.3 Batch VSS

One of the techniques underlying our fast D-PRBG is of independent interest. This is batch Verifiable Secret Sharing (VSS), where we do many verifiable sharings of secrets with a small amortized cost per sharing.

VERIFIABLE SECRET SHARING. The Verifiable Secret Sharing (VSS) problem [10] can be roughly stated as follows. In a distributed system with $n$ players, $P_1, ..., P_n$, $t$ of which may be arbitrarily faulty, there is a special player, the *dealer* $D$, who has a secret he would like to share among the players in a reconstructible manner. The most common way of achieving this is to employ the secret sharing scheme proposed by Shamir [18], in which the secret is the value of a polynomial at the origin, while the players' shares are the values of the polynomial evaluated at the players' id's. Since the dealer can behave arbitrarily, the players need to verify the existence of a valid interpolating polynomial in order to confirm that the dealer has carried out a proper sharing. Furthermore, while determining whether there exists an appropriate polynomial, there is a need to keep the values held by the players secret until the secret is reconstructed. History and existing implementations are discussed in more detail in Section 1.4.

BATCH VERIFICATIONS. The problem we consider is to efficiently verify the distribution of many secrets. We draw from the ideas of batch instance verification [3], whose main goal is to be more efficient than the naive way of verifying each instance individually. Our protocol for batch VSS allows for the verification of multiple secrets *at the same cost of one polynomial interpolation* and in particular much faster, in terms of amortized cost per secret, than any known method.

VERIFICATION OF A SINGLE SECRET. In fact, our techniques also yield a slight improvement for the basic problem of a single (not batch) VSS for a secret of size $k$. Its computation cost is $n^2k$ operations per player, and the communication is $2n$ messages each of size $k$ for a total of $2nk$ bits of communication. Our solution is comparable to the one of [9] in computation, although slightly better in communication. It assumes the existence of a $k$-ary secret coin; this is a realistic assumption in the presence of a D-PRBG, and in particular under the "bootstrapping" setting we are considering here.

### 1.4 History and comparisons

SHARED COINS. The concept of a shared coin was introduced by Rabin [17], who also gave the first implementation. In that setting, the common coin was pre-generated by a trusted party, and could be revealed at a later time by the participants.

There have been many protocols in the literature for the generation of distributed random coins; we shall discuss a few of them. In [11] a coin is achieved which can tolerate $n/\log n$ faults, it runs in constant expected time, and has an additional drawback in that not all the players "see" the coin. Feldman and Micali's protocol [14] is resilient against a third of the players, the computations comprise $O(n^4 \log^2 n)$ steps per player, the communication is $O(n^5)$ messages, and there exists a non-negligible probability that not all players will see the coin. The global coin protocol of Beaver and So [2] only needs a majority a good players, but relies on complexity assumptions (specifically, the intractability of factoring), which in turn makes it inefficient. Furthermore, the generation of bits is limited to a pre-set size.

In comparison, recall that our protocol, assuming a security parameter $k$, will generate $M$ $k$-ary coins and require an amortized computation of $O(n^2 \log k)$ per coin and amortized communication of $O(n)$ messages. In addition, our coin will be unanimous except for a probability of error less than $Mn2^{-k}$. And the generation process is endless, as bits are generated upon demand. As in [2], our scheme also provides "random access" to the bits.

Finally, Blum [6] considered the case of $n = 2$ players, whereas our focus is on $n \geq 4$.

VSS. The verifiability in [10] was implemented via zero-knowledge proofs and thus was expensive. This paradigm continued in several later works. We skip to more direct methods.

Assuming a broadcast channel and a security parameter $k$, the method of Chaum, Crépeau, and Damgård [9] requires $n^2 k \log^2 n$ computation and communication (total number of bits) of $O(nk \log n)$. Their protocol has a probability of error of $2^{-k}$. The VSS protocol of Ben-Or, Goldwasser, and Wigderson [4] is less efficient than this.

Feldman's protocol [12] relies on the unproven assumption of the hardness of discrete log. Assuming a large prime $p$ (length 1024 bits), he achieves $O(n)$ communication and $O(n^2 \log^3 p)$ computation.

## 2 Model

In this paper we consider a synchronous network of $n$ players $P_1, ..., P_n$ (probabilistic polynomial-time machines with a source of perfectly random bits), which communicate by sending messages. We assume that private channels are available between the players. Of the $n$ players, a subset of size at most $t$ of them is assumed to be able to deviate arbitrarily from the protocol, and even collude. We will refer to these players as *faulty*, and to the rest—those who obediently follow the protocol—as *honest* players. This subset of faulty players is only assumed to remain fixed for a constant number of rounds. In this paper we assume $n = O(t)$ (specifically, $n \geq 3t+1$ Section 3, and $n \geq 6t+1$ in Section 4). For Section 3 we assume that a broadcast channel facility is in place; we will show in Section 4 how this assumption can be replaced by point-to-point communication.

We will be working over a finite field whose size will be denoted by $p$ (which is not necessarily a prime). We will be measuring the computational effort of the players executing a protocol by the number of additions that they are required to perform. Recall that naive multiplication in a field of size $2^k$ takes $O(k^2)$ steps. The computation time estimates stated in our results are obtained by working over a specially constructed finite field in which we can multiply faster. Specifically, we can build a field of size $p = \Theta(2^k)$ in which multiplication take only $O(k \log k)$ time. This is done as follows. Let $q$ be a prime and $l$ an integer such that $q \geq 2l+1$ and $q^l \geq 2^k$. We work over $\mathrm{GF}(q^l)$. We view the field elements as degree $l$ polynomials over $Z_q$. Then we use discrete Fourier transforms to do the multiplication, modulo some irreducible polynomial, in $O(l \log l)$ operations over $Z_q$. We can implement operations over $Z_q$ via a table, so that they take $O(\log q)$ time. Choosing $q = O(l)$ and $l = O(k/\log k)$ so that the above constraints are met we end up with a $O(l \log^2 l) = O(k \log k)$ time algorithm. Details of this construction are omitted.

For simplicity however the algorithms we provide below assume we work over $\mathrm{GF}(2^k)$. (But computation time results are stated assuming multiplication takes $O(k \log k)$ time.) We let $p$ denote the field size. We omit the description of the slight changes required to work over the slightly more complex field discussed above.

We also note that in practice, when $k$ is small, working over $\mathrm{GF}(2^k)$ with the naive $O(k^2)$ multiplication is faster than working over our special field with the $O(k \log k)$ multiplication, because of the sizes of the constants involved. So an implementation should be careful about which method it uses.

In some parts we consider the interpolation of a polynomial as a basic step. Methods such as the Berlekamp-Welch decoder [5] can be used to implement this operation.

## 3 Batch Verifiable Secret Sharing

We start off with a VSS protocol for a single secret that compares favorably with existing protocols. This is the first step towards the first major goal of the paper, which is the verification of multiple secrets.

### 3.1 Checking the degree of a polynomial

As mentioned earlier, VSS is intimately related to checking the degree of a polynomial. The problem of checking the degree can be stated as follows. Given values $\alpha_1, ..., \alpha_n$, find out whether there exists a polynomial $f(x)$ such that $deg(f) \leq t$, and $\forall i \in \{1, ..., n\}$, $f(i) = \alpha_i$. The basic solution to this problem is to choose any $t+1$ values (points), without loss of generality $\alpha_1, ..., \alpha_{t+1}$, and to compute the unique polynomial, $f(x)$, that they define (using, say, the Lagrange method). For the remaining points, $\alpha_{t+2}, ..., \alpha_n$, simply check whether they satisfy that $f(j) = \alpha_j$, $t+2 \leq j \leq n$.

In the case of VSS, however, while determining whether there exists an appropriate polynomial, there is a need to keep the values held by the players secret until the secret is reconstructed. This requirement preempts the option of handing out all the values to all the players, and have each of them compute the answer.

We now state the problem more formally.

> **Problem 1 (VSS)** In a system with $n$ players $P_1, ..., P_n$ holding values $\alpha_1, ..., \alpha_n$, respectively, given out by a designated player $D$, determine whether $\exists f(x)$ such that $deg(f) \leq t$, and $\forall i \in \{1, ..., n\}, f(i) = \alpha_i$, while maintaining the values $\alpha_1, ..., \alpha_n$ secret.

It seems that it would be impossible to grant that all the $n$ players' shares will satisfy the polynomial, as some of them might be faulty. Yet it is easy to see that two rounds of broadcast render this possible. However, if the broadcast facility is eliminated, then the players can only check that at most $n - t$ of the shares satisfy the requirements, and then special attention needs to be given to the number of players that are needed in order to reconstruct the secret.

The commonly known solutions to this problem are given in [9, 12, 4]. The method presented in [9] is a *cut-and-choose* protocol. Roughly speaking, the dealer who shared the secret is asked to share $k$ additional polynomials, $g_1(x), \ldots, g_k(x)$. For each $j$, $1 \leq j \leq k$, the players decide whether to reconstruct $g_j(x)$ or $f(x) + g_j(x)$, and check if the reconstructed polynomial is of degree $\leq t$. Thus, in this approach $k$ polynomial interpolations are computed in order to achieve a probability of error less than $\frac{1}{2^k}$.

Feldman's protocol [12] depends on the unproven assumption of the hardness of the discrete log problem. After defining the polynomial (*à la* Shamir) and computing all the private shares $f(i)$ of the players, the dealer generates public information which aids in the verification. A consequence of this is that both the dealer and the players have to carry out $t$ exponentiations (i.e., $t \log p$ multiplications). See [9, 12] for details.

The VSS protocol we present here computes a single polynomial interpolation in a field GF$(2^k)$, and achieves a probability of error less than $\frac{1}{2^k}$. Note that this achieves the same probability of error as in [9]. The number of required computations is $2n^2k$, and the communication required by our protocol is $2n$ messages, each of size $k$, resulting in a total bit complexity of $2nk$.

Protocol `VSS` is shown in Figure 2. The code is self-explanatory. We are assuming that the players have access to a secret random *k-ary-coin*. `Coin-Expose` is the protocol which is executed in order to reveal the value of the secretly-held coin. We give an implementation for such a protocol in Section 4, under the assumption that the secret coin was generated by our protocol. As all our coins will be generated in the field GF$(2^k)$ we can assume that each coin generates in fact $k$ random coins in $\{0, 1\}$. Hence, we shall call these coins "$k$-coins." The random $k$-ary coin from Step 2 is used as a scalar $< 2^k$. For short, we will say that protocol `VSS` accepts (or rejects), meaning that all players output "accept" (or "reject"). We now address the correctness and efficiency of the VSS protocol.

**Lemma 1** *Assume $\neg\exists f(x)$ such that $deg(f) \leq t$ and $f(i) = \alpha_i$, $\forall i \in \{1, ..., n\}$. Then protocol* `VSS` *accepts with probability at most $\frac{1}{p}$.*

**Proof** The shares $\alpha_1, ..., \alpha_n$ interpolate a single polynomial $f'(x)$ of degree at most $n - 1$. Due to the assumption, $deg(f') \geq t + 1$. Assume that $f'$ is of the form $f'(x) = a_{n-1}x^{n-1} + ... + a_1 x + a_0$, where there exists a $j \geq t + 1$ such that $a_j \neq 0$. In order for `VSS` to accept it must be the case that $F(x)$, the polynomial computed by all players in Step 4, is of degree at most $t$, hence the coefficient of $x^j$ in $F(x)$ is zero. A necessary requirement for this to happen is that the coefficient of the term $x^j$ of $g(x)$ equals $-a_j/r$. As $g(x)$ was determined before exposing $r$, a malicious dealer has a probability of at most $\frac{1}{p}$ of setting the coefficient to the appropriate value. □

**Protocol** `VSS`

**Given:**

- A system with $n$ players $P_1, ..., P_n$ holding values $\alpha_1, ..., \alpha_n$, respectively, given out by a designated player $D$.
- Access to a secret random $k$-ary-coin.

**Check:** Whether there exists a polynomial $f(x)$ s.t. $deg(f) \leq t$, and $\forall i,\ f(i) = \alpha_i$,

1. The dealer $D$ shares an additional polynomial $g(x)$, so that each player $P_i$ has a share $\beta_i \stackrel{\triangle}{=} g(i)$.
2. Set $r \leftarrow$ `Coin-Expose`($k$-ary-coin).
3. Player $P_i$ computes $\gamma_i \stackrel{\triangle}{=} \alpha_i + r\beta_i$, and broadcasts this value.
4. Interpolate a polynomial $F(x)$ through $\gamma_1, ..., \gamma_n$. If $deg(F) \leq t$ output "accept," otherwise output "reject."

Figure 2: An efficient protocol for VSS.

**Lemma 2** *For the verification of one secret, protocol* `VSS` *requires* $n + k \log k + 1$ *additions and 2 polynomial interpolations per player. There are 2 rounds of communication, and the number of messages in each round is* $n$*, each of size* $k$*, for a total of* $2nk$ *bits.*

The proof of this lemma follows immediately from inspection of the protocol. We note that a single secret coin is reconstructed for the verification of the secret. The protocol we present for this purpose (`Coin-Expose`, Section 4) requires $n$ additions and a single interpolation of a polynomial per player. And the communication it requires is $n$ messages, each of size $k$. Hence, it is equivalent in computation to the interpolation of the shares being examined.[1]

We now turn to the batch verification of degrees of polynomials and VSS instances.

### 3.2 Batch checking of degree of polynomials

The batch checking for the problem from the previous section can be stated as follows.

> **Problem 2 (Batch VSS)** In a system with $n$ players $P_1, ..., P_n$, where player $P_i$ holds the values $\alpha_{i1}, ..., \alpha_{iM}$, given out by a designated player $D$, determine whether there exist polynomials $f_1(x), ..., f_M(x)$ such that $\forall j$ and $\forall i$, $deg(f_j) \leq t$ and $f_j(i) = \alpha_{ij}$, while maintaining the values $\alpha_{i1}, ..., \alpha_{iM}$ secret for all $i$.

We now give an efficient solution to this problem. Again we shall assume the existence of a $k$-ary-coin (i.e., a set of distributed random bits which are secret in the initial phase when the shares are being given out to the players). The $k$-ary-coin will be utilized to compute a linear

[1]Nonetheless, the computation for exposing a coin depends on the specific implementation of the coin.

**Protocol** `Batch-VSS`

**Given:**

- A system with $n$ players $P_1, ..., P_n$ where player $P_i$ holds values $\alpha_{i1}, ..., \alpha_{iM}$.
- Access to a secret random $k$-ary-coin.

**Check:** Whether there exist polynomials $f_1(x), ..., f_M(x)$ s.t. $\forall j$ and $\forall i$, $deg(f_j) \leq t$ and $f_j(i) = \alpha_{ij}$.

1. Set $r \leftarrow$ `Coin-Expose`($k$-ary-coin).
2. Player $P_i$ computes $\beta_i \triangleq r^M \alpha_{iM} + \ldots r\alpha_{i1}$. (This can be efficiently computed as $(\cdots((r\alpha_{iM} + \alpha_{i(M-1)})r + \alpha_{i(M-2)})\cdots)r + \alpha_{i1})r$. )
3. Player $P_i$ broadcasts $\beta_i$.
4. Interpolate a polynomial $F(x)$ through $\beta_1, ..., \beta_n$. If $deg(F) \leq t$ output "accept," otherwise output "reject." (The interpolation can be done by a single player and verified by the rest.)

Figure 3: Batch verification of degree of polynomials.

combination $F(x)$ of all the polynomials. Then, in a single interpolation, we will check whether the polynomial $F(x)$ is of degree at most $t$. This will yield a slightly different probability of error than the one from protocol `VSS` . The protocol is shown in Figure 3.

The protocol of Figure 3 can be easily modified to "accept" if there is a polynomial $F(x)$ of degree at most $t$, which for some given $l$, satisfies that for values $\beta_{i_1}, ..., \beta_{i_l}$ we have $F(i_j) = \beta_{i_j}$, for $j \in \{1, .., l\}$. We will denote the execution of the above protocol with argument $l$ as `Batch-VSS`$(l)$.

**Lemma 3** *Assume $\exists f_j(x)$ such that $deg(f_j) > t$ and $\forall i \in \{1, ..., n\}$, $f_j(i) = \alpha_i$. Then protocol* `Batch-VSS` *accepts with probability at most $\frac{M}{p}$.*

**Proof** Given a polynomial $f_j(x) = a_m x^m + ... + a_1 x + a_0$, where $a_m \neq 0$, denote by

$$f_j(x)|^{t+1} \triangleq a_m x^m + ... + a_{t+1} x^{t+1}.$$

If $m \leq t$, then $f_j(x)|^{t+1} = 0$. In order for `Batch-VSS` to output "accept," it must be that $deg(F) \leq t$. Thus, $\sum_{j=1}^{M} r^j f_j(x)|^{t+1}$ must be equal to 0. This is an equation of degree $M$ and hence has at most $M$ roots. In order for `Batch-VSS` to fail, the value of $r$ must be equal to one of the roots of the equation. However, this can happen with probability at most $\frac{M}{p}$. □

**Lemma 4** *For the verification of $M$ secrets with security parameter $k$, protocol* `Batch-VSS` *requires $2Mk\log k$ additions and a 2 polynomial interpolations per player. There are two rounds of communication, each with $n$ messages. The size of the messages is $k$ in the first round, $k$ in the second, for a total of $2nk$ bits.*

The lemma can be immediately corroborated from inspection of the protocol. Note that a *single* coin is reconstructed for the verification of multiple secrets, and again that the complexity of exposing the coin is equivalent to that of interpolating the polynomial for the verification. Lemma 4 yields the following amortized cost for secret verification.

**Corollary 1** *Using* `Batch-VSS`, *the amortized computation required to verify a secret is* $2k \log k$ *per player, and the amortized communication is* $O(1)$.

## 4 A Bootstrap Method for Distributed Coin Generation

Now that we have shown that we can efficiently check a bulk instance of secret sharings, we can exploit this to generate shared secret random coins in an efficient manner. In a distributed system which might incur faults, one cannot rely on one processor to generate a coin, but rather need to have a communal effort to generate random bits; a random coin is then achieved by somehow combining (e.g., XOR-ing) the individual bits. Coins are often used as a source of randomness to execute Byzantine agreement, and hence implement a broadcast channel. Thus, we will omit the assumption of a broadcast channel from the model. Yet, if the coins are used for an application other than broadcast, then the simpler algorithm which assumes broadcast can be utilized.

A straightforward way to generate a coin would be to interpolate a number of polynomials which at least equals the number of the faults to be tolerated. Coins generated this way, however, would still be highly expensive. In this section we show how to achieve this with just one polynomial interpolation.

In this section we assume $n \geq 6t + 1$ and the availability of secret random $k$-ary-coins.

We introduce protocol `Bit-Gen` which will employ a modified version of the `Batch-VSS` protocol in order to conform to the requirement of point-to-point communication. `Bit-Gen` enables a dealer to share $M$ secrets, while allowing the players to verify that the dealer has shared proper secrets. The protocol is shown in Fig. 4. We remark that a run of `Bit-Gen` generates several (i.e., $M$) bits. As there is no broadcast channel, every time a player needs to announce a message, (s)he can only distribute it to each of the other players individually. As a result, there can be no agreement among the players on whether the `Bit-Gen` execution of some player has been completed successfully, but rather each player can only reach a local decision. In order to generate the coin we will need a consistent view, i.e., that all the honest players agree on which of the `Bit-Gen` protocols have been completed successfully. We will deal with this problem in the sequel. First, we state the correctness and complexity of the `Bit-Gen` protocol.

**Lemma 5** *Assume* $\exists f_j(x)$ *such that* $deg(f_j) > t$, *and that for a subset of size at least* $n - 2t$ *of honest players* $i$ *it holds that* $f_j(i) = \alpha_i$. *Then an honest player executing* `Bit-Gen` *outputs* $(F(x), S)$ *(namely, accepts the sharing) with probability at most* $\frac{M}{p}$.

The proof of this lemma is similar to the proof of Lemma 3, and omitted from this abstract.

**Lemma 6** *For the generation of* $M$ *shared secrets with security parameter* $k$, *protocol* `Bit-Gen` *requires* $Mtk \log k + 2Mk \log k$ *additions and 2 polynomial interpolations per player. There are 3 rounds of communication. In the first round there are* $n$ *messages each of size* $Mk$, *in the second and third rounds* $n^2$ *messages of size* $k$, *for a total of* $nMk + 2n^2k$ *bits.*

As the secrets which are shared are numbers in $Z_p$, we can regard each one as $k$ bits in $\{0,1\}$. Thus we achieve:

**Protocol** `Bit-Gen`

**Given:**

- Access to a secret random $k$-ary-coin.

**Protocol for Dealer:**

1. Define
$$f_1(x) = f_{1,0} + f_{1,1}x + \ldots + f_{1,t}x^t$$
$$\vdots$$
$$f_M(x) = f_{M,0} + f_{M,1}x + \ldots + f_{M,t}x^t$$
Compute $f_j(i)$ for $1 \leq j \leq M \quad 1 \leq i \leq n$, and send to player $P_i$ the values $f_j(i)$.

**Protocol for player $P_i$ upon receiving values $\alpha_{i1}, \ldots, \alpha_{iM}$:**

2. Set $r \leftarrow$ `Coin-Expose`($k$-ary-coin).

3. Compute $\beta_i \triangleq (\cdots((r\alpha_{iM} + \alpha_{i(M-1)})r + \alpha_{i(M-2)})\cdots)r + \alpha_{i1})r$, and send $\beta_i$ to all players.

4. Set $S \leftarrow \{\beta i_1, \ldots, \beta_{i_j}\}$, which were received in the previous step.

5. Using the Berlekamp-Welch decoder, interpolate a polynomial $F(x)$ through the shares in $S$. If there exists a polynomial $F(x)$ s.t. $deg(F) \leq t$ and for at least $n - t$ values in $S$ it holds that $F(i_j) = \beta_{i_j}$, then output $(F(x),\ S)$, otherwise output$(\perp,\ S)$.

Figure 4: Generation of sealed bits.

**Corollary 2** *Using* `Bit-Gen`*, the amortized computation required to generate a single bit in* $\{0,1\}$ *is* $n \log k + O(\log k)$*, and the amortized communication is* $n + O(1)$*.*

We now turn to the generation of shared coins. The protocol incorporates the `Bit-Gen` protocol, and then lets the players determine which of the protocols have been completed successfully. More specifically, our approach is as follows: each player executes `Bit-Gen` as the dealer, and has a local opinion on how each one of the protocols terminated. The idea is that players then execute a single Byzantine agreement protocol in order to decide which of the `Bit-Gen` protocols were completed properly. This, however, cannot be done directly on the output of the `Bit-Gen` protocols, due to the following. A successful execution of `Bit-Gen` guarantees that the bits can be eventually reconstructed and known to the whole group, yet the information which enables the reconstruction is available, in the worst case, to only a subset of the players. This is true for each of the `Bit-Gen` protocols, hence the subsets, most likely, will not be the same. If in generating the coin we would need to generate each bit individually there would be a high overhead in computation. Hence, we would like to ensure that there will exist a single subset which is capable of reconstructing all the bits, and in turn make the computation of the coins much more efficient (namely, by executing a single interpolation to generate a coin).

We therefore apply an efficient method for finding a clique of a certain size within the graph. Intuitively, our problem generates the following directed graph $\bar{G}(V', E')$. Each player represents a node (hence $|V'| = n$), and a directed edge from $j$ to $k$ is in $E'$ if $P_k$ has a proper share of the bits which $P_j$ shared. We now define the graph $G(V, E)$ as follows: set $V$ to $V'$, and add an edge $(j, k)$ to $E$ if both $(j, k)$ and $(k, j)$ are in $E'$. Let us observe the properties of $G(V, E)$. For every honest players $P_i$ and $P_j$, the edge $(j, k) \in E$. It is also easily seen that if the dealer in `Bit-Gen` is honest, then each honest player has a proper share of the bits (including the dealer). Due to the above, there is a clique of size at least $n - t$ in $G$. Utilizing the protocol of Gabril ([15], p. 134), a clique can be found of size at least $n - 2t$. This clique can be computed locally by each player, and then distributed. Once this is done, the players need to carry out a *single* Byzantine agreement to decide on one of these cliques.

The protocol for the generation of shared coins, `Coin-Gen`, is shown in Fig. 5. In the protocol, `Grade-Cast` is the three level-outcome primitive used in his/her value to the rest of the players. In the next round everybody echoes, and this is followed by another round of echos. Each player outputs a value $\sigma$, which is the view of the grade-casted message, and a confidence value *conf* $\in \{0, 1, 2\}$ indicating how certain (s)he is that the grade-cast was received by all players. A confidence of 2 indicates that all other honest players have seen the value $\sigma$. See [14] for details.

We now turn to arguing the correctness of `Coin-Gen`. For simplicity, we assume that $n = 6t + 1$. Recall that, upon termination, the output of the protocol consists of a set of players.

**Lemma 7** *Assume honest player $P_i$ outputs a set $U_i$ at the end of* `Coin-Gen`. *Then the following hold:*

1. $|U_i| \geq n - 2t + 1 = 4t + 1$;
2. *for all honest players $j$, $U_i = U_j$; and*
3. *there exists a subset $U \subset U_i$, $|U| \geq 2t + 1$, of honest players such that each player $P_j \in U$ has shares of the secrets of each player $P_k \in U_i$.*

**Proof** [Sketch] Protocol `Coin-Gen` has terminated indicating that the `BA` protocol has terminated with a value of 1. This in turn implies that at least one honest player has run `BA` with value 1 as the input (otherwise, the Validity condition of Byzantine agreement would not be satisfied), and hence satisfied conditions i)-iii) of Step 10. Due to condition i) we have that all the honest players' view of $C_l$ is the same, and due to ii), that $|C_l| \geq 4t + 1$, establishing 1 and 2. The honest player who inputted 1 into `BA` checked that at least $3t + 1$ players have shares of all the other players' secret, and hence there is a set of size at least size $2t + 1$ of honest players who satisfy this property. This establishes 3. □

**Lemma 8** *Protocol* `Coin-Gen` *terminates in constant expected time.*

**Proof** [Sketch] The protocol re-iterates `BA` only if the previous execution has ended with a 0 outcome. This can happen only if $P_l$ is faulty. As the faulty players are set before $l$ is exposed, there is a probability of at least $n - \frac{t}{n}$ that `BA` will terminate with a value of 1. □

**Theorem 1** *Protocol* `Coin-Gen`, *in conjunction with* `Coin-Expose`, *enables the generation of $M$ shared random coins.*

**Proof** [Sketch] We are guaranteed that since at most $t$ of the players are faulty, at least $2t+1$ players in $S$, the set of the chosen clique members, have proper shares of the coin. This enables us to use the Berlekamp-Welch [5] decoder to compute the desired polynomial. $\square$

The following theorem states the cost of generating the shared coins.

**Theorem 2** *Protocol* `Coin-Gen` *requires:*

- Computation: *The $n$ parallel executions of* `Bit-Gen` *require $Mn^2k\log k + 2Mnk\log k$ additions, and $n+1$ interpolation per player.* `Bit-Gen`.*In addition, each player needs to compute a clique, and an expected constant number of interpolations and* `BA`*s.*
- Communication: *In the $n$ parallel executions of* `Bit-Gen` *there are $n$ messages of size $Mnk$, and $n^2$ messages of size $kn$. Step 6 requires $n^2$ messages each of size $ntk$. In addition, Step 8 requires $n^2$ messages of size $k$ and the cost of* `BA`*, for a total of $Mn^2k + O(n^4k)$ bits.*

Regarding computation costs, note that $n$ polynomial interpolations have been saved by using the same coin for all the invocations of `Bit-Gen`. Theorem 2 yields:

**Corollary 3** *Given that $M$ coins have been generated, each of size $k$, the amortized cost of computation per coin $\in \{0,1\}$ is $O(n^2\log k)$ operations, and the amortized communication is $n^2 + O(n^4/M)$ bits.*

Note that each coin needs a separate interpolation, and this can not be amortized.

## 5 Summary

In this paper we have introduced a new paradigm, D-PRBGs, for efficiently generating shared random coins in a distributed system, with low amortized cost per coin produced. This cost (computation and communication) is significantly lower than the cost of any "from-scratch" shared coin generation protocol. As the bottleneck for distributed coin generation in such a setting is the final interpolation of the coin, the amortized cost of our method does not exceed this value.

We also showed how to bootstrap our coin generation method in order to generate an unlimited number of coins. This construction of bootstrapping fits nicely in applications that require proactiveness.

## Acknowledgments

We thank Hugo Krawczyk and Moti Yung for helpful comments on a preliminary version of this manuscript. We also thank the members of the PODC committee for their comments.

## References

[1] A. Bar-Noy, X. Deng, J. Garay, , and T. Kameda. Optimal Amortized Distributed Consensus . In *Proc. 5th International Workshop on Distributed Algorithms*, volume 579 of *Lecture notes in computer Science*, pages 95–107. Springer-Verlag, 1991.

[2] D. Beaver and N. So. Global, Unpredictable Bit Generation without Broadcast. In *Advances in Cryptology – Eurocrypt 93*, volume 765 of *Lecture Notes in Computer Science*, pages 424–434. Springer-Verlag, 1993.

[3] M. Bellare, J. Garay, and T. Rabin. Fast Bulk Verification of Discrete Logs and DSA and RSA Signatures, 1996. In preparation.

[4] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Noncryptographic Fault-Tolerant Distributed Computations. In *Proceeding 20th Annual Symposium on the Theory of Computing*, pages 1–10. ACM, 1988.

[5] E. Berlekamp and L. Welch. Error correction of algebraic block codes. US Patent 4,633,470.

[6] M. Blum. Coin Flipping by Telephone– A protocol for solving impossible problems. In *IEEE Spring Compcon*, pages 133–137. IEEE, 1982.

[7] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13:850–864, 1984.

[8] R. Canetti and A. Herzberg. Maintaining security in the presence of transient faults . In *Advances in Cryptology - CRYPTO '94 proceeding*, volume 839 of *Lecture notes in computer Science*, pages 425–438. Springer-Verlag, 1994.

[9] D. Chaum, C. Crépeau, and I. Damgå rd. Multiparty Unconditionally Secure Protocols. In *Proceeding 20th Annual Symposium on the Theory of Computing*, pages 11–19. ACM, 1988.

[10] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults. In *Proceeding 26th Annual Symposium on the Foundations of Computer Science*, pages 383–395. IEEE, 1985.

[11] C. Dwork, D. Shmoys, and L. Stockmayer. Fliping Presuasively in Constant Expected Time. In *Proceeding 27th Annual Symposium on the Foundations of Computer Science*, pages 222–232. IEEE, 1986.

[12] P. Feldman. A Practical Scheme for Non-Interactive Verifiable Secret Sharing. In *Proceeding 28th Annual Symposium on the Foundations of Computer Science*, pages 427–437. IEEE, 1987.

[13] P. Felman and S. Micali. Byzantine Agreement in Constant Expected Time (and trusting no one). In *Proceeding 26th Annual Symposium on the Foundations of Computer Science*, pages 267–276. IEEE, 1985.

[14] P. Felman and S. Micali. An Optimal Algorithm for Synchronous Byzantine Agreement . In *Proceeding 20th Annual Symposium on the Theory of Computing*, pages 148–161. ACM, 1988.

[15] M. R. Garey and D. S.Johnson. *Computers and Intractability: a guide to NP-Completeness.* W. H. Freeman, ed. N.Y., 1979.

[16] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive Secret Sharing. In *Advances in Cryptology - CRYPTO '95 proceeding*, volume 963 of *Lecture notes in computer Science*, pages 339–352. Springer-Verlag, 1995.

[17] M. Rabin. Randomized Byzantine Generals. In *Proceeding 24th Annual Symposium on the Foundations of Computer Science*, pages 403–409. IEEE, 1983.

[18] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22:612–613, 1979.

[19] A. Yao. Theory and applications of trapdoor functions. In *Proceeding 27th Annual Symposium on the Foundations of Computer Science*. IEEE, 1986.

**Protocol** `Coin-Gen`

**Given:**

- Access to $O(1)$ secret random $k$-ary-coins.

**Protocol for player $P_i$:**

1. As Dealer, initiate $\texttt{Bit-Gen}_i$ Step 1 (Fig. 4).
2. Set $r \leftarrow$ `Coin-Expose`($k$-ary-coin).
3. Participate in all invocations of $\texttt{Bit-Gen}_j$ Steps 3-5, $1 \leq j \leq n$, using the same coin $r$ for all invocations. set local values $F_j$ and $S_j$ to the output of $\texttt{Bit-Gen}_j$.
4. Define a directed graph $\vec{G}(V', E')$ in the following manner: Let each player denote a vertex in the graph. Add a directed edge from $j$ to $k$ if $F_j \neq \perp$ and $P_k$'s share $\beta_k$ is in $S_j$ and satisfies that $F_j(k) = \beta_k$.
5. Define the graph $G(V, E)$ as follows: Set $V$ to $V'$; add an edge $(j, k)$ to $E$ if both $(j, k)$ and $(k, j)$ are in $E'$.
6. Find a clique $C$ of size at least $n - 2t \geq 4t + 1$ in $G$.
7. $\texttt{Grade-Cast}_i(\{(j, F_j)\})$ s.t. $j \in C$.
8. Set $C_j$ to the value received from player $P_j$, And $conf_j$ to the confidence of the $\texttt{Grade-Cast}_j$ on this value.
9. Set $l \leftarrow$ `Coin-Expose`($k$-ary-coin) mod $n$. (If $l = 0$ then set $l \leftarrow n$).
10. Run any `BA` protocol. $P_i$'s input to the protocol is 1 if:

    i) $conf_l = 2$;

    ii) $|C_l| \geq 4t + 1$; and

    iii) there exist $\geq 3t + 1$ $j$'s in $C_l$ s.t. for all other $k \in C_l$ it holds $\beta_j \in S_k$ satisfies the polynomial $F_k$.

    Otherwise input is $\perp$.
11. If `BA` terminates with 1, then set *output* to $C_l$. Otherwise, repeat from Step 9.

Figure 5: Generation of sealed coins.

**Protocol** `Coin-Expose`

**Given:**

- A set $S \triangleq \{P_1, ..., P_{3t+1}\}$ (wlog) of players (subset of clique members, which satisfied condition iii) in previous run of `Coin-Gen`).
- The number $h$ of the coin to be exposed.

**Protocol for player $P_i$:**

1. For $P_i \in S$, compute $\alpha_i \triangleq \sum_{j=1}^{3t+1} \alpha_{i,j,h}$, where $\alpha_{i,j,h}$ is the $h$-th share received from player $P_j$. Send $\alpha_i$ to all players.
2. Using the Berlekamp-Welch decoder, interpolate a polynomial $F(x)$ through the shares received in the previous step.
3. Set $coin_h = F(0) \bmod 2$.

Figure 6: Exposing a coin.