RandSolomon: Optimally Resilient Random Number Generator with Deterministic Termination

Luciano Freitas de Souza
CEA LIST, Université de Paris-Saclay, Gif-sur-Yvette, France
LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Andrei Tonkikh
LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Sara Tucci-Piergiovanni
CEA LIST, Université de Paris-Saclay, Gif-sur-Yvette, France

Renaud Sirdey
CEA LIST, Université de Paris-Saclay, Gif-sur-Yvette, France

Oana Stan
CEA LIST, Université de Paris-Saclay, Gif-sur-Yvette, France

Nicolas Quero
CEA LIST, Université de Paris-Saclay, Gif-sur-Yvette, France

Petr Kuznetsov
LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Abstract

Multi-party random number generation is a key building-block in many practical protocols. While straightforward to solve when all parties are trusted to behave correctly, the problem becomes much more difficult in the presence of faults. This paper presents RandSolomon, a partially synchronous protocol that allows a system of N processes to produce an unpredictable common random number shared by correct participants. The protocol is optimally resilient, as it allows up to $f = \lfloor \frac{N-1}{3} \rfloor$ of the processes to behave arbitrarily, ensures deterministic termination and, contrary to prior solutions, does not, at any point, expect faulty processes to be responsive.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Byzantine Fault Tolerance, Partially Synchronous, Deterministic Termination, Randomness Beacon, Multi Party Computation, BFT-RNG

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2021.23

Funding Luciano Freitas de Souza was supported by Nomadic Labs, Petr Kuznetsov and Andrei Tonkikh – by TrustShare Innovation Chair.

1 Introduction

In a Byzantine fault-tolerant random number generator (BFT-RNG) protocol, a set of participating processes agree on a single random number that cannot be manipulated or halted, despite the presence of Byzantine failures, i.e., assuming that a faulty process may arbitrarily deviate from the prescribed algorithm. We distinguish between *commission* and *omission* failures [15]. Intuitively, a *commission* fault occurs when a process sends messages a correct process would not send, whereas an *omission* fault occurs when a process does not send messages a correct process would send.

A BFT-RNG protocol is typically divided into three phases:

1. **Generation and Commitment Phase** – each process locally generates some random value and then publicly commits to this value without revealing it.
2. **Reveal Phase** – the values previously committed are revealed.

3. **Computation Phase** – using the values revealed, the processes decide on the resulting random number.

The idea is to make sure that at the moment the committed random values are revealed, it is already too late for the adversary to manipulate the output. Furthermore, assuming that the local random numbers are uniformly distributed, so should be the distribution of the output.

To the best of our knowledge, this paper describes the first partially synchronous BFT-RNG protocol that maintains optimal resilience (up to $\lfloor \frac{N-1}{3} \rfloor$ Byzantine processes in a system of N) that ensures *deterministic* termination. Unlike prior solutions [10, 31], our protocol does not expect that faulty processes remain responsive in the generation phase, i.e., it tolerates omission faults.

State of the art. In designing a BFT-RNG algorithm, we face two major challenges: (i) how to share random inputs despite omission failures, so that Byzantine processes cannot learn them before the reveal phase begins, and (ii) how to compute correct results despite commission faults of Byzantine processes. Existing protocols solve the first challenge by using techniques such as secret sharing [28], verifiable delay functions [4], threshold signatures [3, 5], and fully homomorphic encryption [13] and the second – by requiring a verifiable proof that a shared data was generated correctly.

Techniques. A (f, N)-secret sharing [28] scheme allows a process during the generation and commitment phase to share a secret s with N processes so that any subset of size $f+1$ among them can retrieve s, while no subset of f or less can. This way, even if a process refuses to disclose the original secret it has committed, the correct processes in the system can still reconstruct it in the reveal phase by using the shares they received earlier. Moreover, the values cannot be learned too early as the number of shares held by the Byzantine processes does not surpass f. *Threshold-signature* schemes, such as Schnorr [3] or BLS [5], are also very helpful in this context, as they allow to efficiently verify that a number of processes surpassing a given threshold agree with a certain value.

One can also make sure that the processes commit to a value without revealing it beforehand and provide a mechanism to retrieve commitments of Byzantine processes by using *verifiable delay functions* [4]. This technique guarantees that Byzantine processes cannot use the data shared by the correct processes to change change their inputs and affect the result. Once a stipulated verifiable delay has expired, the correct processes can access the information presented by any process guaranteeing that the protocol is not halted.

The two homomorphic structures of most interest for BFT-RNG are Fully Homomorphic Encryption (FHE) [13] and homomorphic hashes. Given two sets A and B, a map $f : A \to B$ is said to be ($\circ$-)homomorphic if it preserves an existing operation $\circ$ on both sets: $\forall x, y \in A, f(x \circ y) = f(x) \circ f(y)$ [6]. FHE allows processes to make operations in ciphertexts without knowing the plaintexts and can be then used instead of secret sharing for solving the same problem of preventing misbehaving parties from accessing data too early on and denying the access of correct participants to the data when it must be shared. As for homomorphic hashes, they are, as the name indicates, hash functions with homomorphic properties (i.e. by performing some operations over some data and their associated hashes, one obtains a result and a consistent associated hash). Homomorphic hashes allow to solve the second challenge of BFT-RNG design: they provide a mean to check that an operation was correctly executed by observing the hashes of the inputs and the hash of the outputs and can therefore contribute in detecting commission failures.

Other kinds of proofs of well formed data include Verifiable Random Functions (VRF) or Public Verifiable Secret Sharing (PVSS). VRF [21] are functions that once provided with an input x, output both a random number y and a proof π that allows any process using π to verify whether y was generated using x or not. Algorand's VRF [14] uses a common coin (generated by the Algorand consensus) to correctly generate verifiable random numbers. PVSS-based proof [27] exchange together with secret shares some additional information that prove the data integrity without revealing any information of the original secret.

Protocols. In Table 1, which is a modified and expanded version of the table given in [26], we present a comparison including several existing BFT-RNG algorithms and the solution we present in this paper: RandSolomon. In some of these protocols, the networks (with N nodes) are partitioned into clusters of size c, this parameter appears in some of the complexity bounds given in the table.

Table 1 Comparison of distributed RNG solutions.

RNG	Sync.	Vulnerability	Term.	Communication Complexity (Overall)	Computation Complexity (per process)	Resilience	Techniques
Cachin et al. [10]	A	Trusted key dealer	Det.	$O(N^2)$	$O(N)$	$f < \frac{N}{3}$	Unique threshold signatures (eg BLS)[5]
RandShare [31]	A	No ommission in commit.	Det.	$O(N^3)$	$O(N^3)$	$f < \frac{N}{3}$	PVSS [27]
RandHound[31]	A	No ommission in commit.	Prob.	$O(c^2N)$	$O(c^2N)$	$f < \frac{N}{3}$	PVSS [27] Multisignatures [3]
RandHerd[31]	A	No ommission in commit.	Prob.	$O(c^2 \log N)$	$O(c^2 \log N)$	$f < \frac{N}{3}$	PVSS [27] Multisignatures [3]
SCRAPE[11]	S	None	Det.	$O(N^3)$	$O(N^2)$	$f < \frac{N}{2}$	PVSS [27]
DFinity[16]	S	None	Prob.	$O(cN)$	$O(c)$	$f < \frac{N}{2}$	BLS signatures [5]
HydRand[26]	S	No ommission in commit.	Det.	$O(N^2)$	$O(N)$	$f < \frac{N}{3}$	PVSS [27]
ProofOfDelay[8]	S	None	Det.	$O(N)$ + Ethereum	High	$f < \frac{N}{2}$	Delay functions [4]
No-Dealer[18]	S	None	Det.	$O(N^2)$	$O(N^2)$	$f < \frac{N}{2}$	Shamir [28] Homomorphic Hash
Nguyen et al.[22]	S	Trusted Requester	Prob.	$O(N)$	$O(1)$	$f < N$	FHE [13], VRF [21]
Ouroboros Praos[12]	P	Weaker properties	Det.	$O(N)$ + Ourob. Praos	$O(1)$ + Ourob. Praos	$f < \frac{N}{3}$	VRF [21]
Algorand[14]	P	Weaker properties	Prob.	$O(cN)$ + Algorand	$O(c)$ + Algorand	$f < \frac{N}{3}$	VRF [21]
RandSolomon	P	None	Det.	$O(N)\times$ Consensus	$O(N)\times$ *Erasure Correcting Code*	$f < \frac{N}{3}$	PK crypto ReedSolomon Retraceability

Synchrony (Sync). The second column of the comparison table shows which kind of synchrony the underlying system must provide in order to allow the deployment of each protocol. Here we distinguish A=Asynchronous, S=Synchronous and P=Partially Synchronous algorithms.

Vulnerability. It might seem impossible to have asynchronous implementations of BFT-RNG as we have already stated that this problem is impossible in the presence of at least one Byzantine participant in asynchronous systems [18]. Notice, one might introduce additional assumptions on the failure model for these solutions to exist.

This is the case with the solution by Cachin et al. [10] which assumes that there exists a special process capable of generating and distributing a key.

Other asynchronous solutions, such as RandShare, RandHound and RandHerd [31], assume that *every* entity initially publishes some information about their secret. The asynchronous protocols in [31] are therefore not fully BFT, as they do not tolerate omission failures in the generation phase. This assumption that Byzantine processes will not omit during the commitment phase of the protocol is also an exploitable vulnerability in the *synchronous* protocol HydRand [26], although it can be modified to restart once there are missing contributions. Nguyen et al.'s proposal [22], also a synchronous protocol, assumes a *Requester*, a trusted entity generating FHE keys, which can be considered as a client using the system.

Algorand [14] and Ouroboros Praos [12], maintain weak forms of RNG: common coin [14] and random beacon [12], RNG mechanisms in these protocols may not reach perfect agreement on the random value, and the coins values may be manipulated by the adversary to some extent or even be changed due to network asynchrony without affecting the correctness of their respective systems.

Termination (Term). A protocol ensures *deterministic termination* (Det) if it terminates in every execution, in contrast to *probabilistic termination* (Prob), when a protocol terminates with a fixed probability. RandHound, RandHerd [31] and Dfinity [16] allow a small probability, depending on the parameters of the system, of the Byzantine adversary fully corrupting a cluster, which results in prematurely halting the protocol. In the case of Algorand, a failure happens when the set (of expected cardinality c) of nodes chosen to be proposers is empty. In the protocol by Nguyen et al. [22], this happens when all selected contributors are Byzantine.

Complexity. Communication complexity corresponds to the amount of messages exchanged and can be loosely translated into how many bits must be sent in the network for producing a result, while *Computation Complexity* measures how much time would it take to perform local computations given an input. In the table, we use term *High* to refer to the complexity of delay functions, which, though independent of the number of processes in the system (strictly speaking, their complexity is $O(1)$), are very computationally heavy by design.

No-Dealer [18] specifies that the protocol must be restarted in case of certain Byzantine behavior, but does not include this fact in its complexity. As there are at most $\frac{N}{2}$ Byzantine nodes, it might be necessary to restart this number of times, increasing their claimed complexity to the one presented in the table.

The protocol by Nguyen et al. [22] employs a summation on the secrets shared by the contributors, which results in linear computation complexity.

Finally, the two last columns **Resilience** and **Techniques** show how many Byzantine processes can be tolerated among the N participants and the main techniques employed in each solution.

Contributions. RandSolomon is the first BFT-RNG protocol providing deterministic termination in a partially synchronous system with $f < \frac{N}{3}$ Byzantine processes, which is the optimal level of resilience [18]. Interestingly, the protocol relies only on standard cryptographic primitives: a public key infrastructure [25], block erasure correcting codes which can be interpreted as our version of secret-sharing [20] and standard digital signatures. The name of the protocol is inspired by the potential use of Reed-Solomon codes [24].

Our coding approach carries some similarities with SCRAPE [11] in the sense that they also recognised the potential of using codes such as Reed-Solomon to perform secret sharing. However the similarities stop there as, in RandSolomon, we not only propose a partially synchronous solution, but also introduce a new technique to cope with Byzantine commission

failures: *retraceability*, which circumvents the need for verification of the secret sharing. In a nutshell, we consider the secrets produced by Byzantine processes without checking their integrity until the last phase of the protocol, when we compute the final result. At this moment, we can retrace all the steps that should have been taken and detect a commission failure. This then results in discarding incorrectly formed data in order to ensure a correct result, based on the inputs of non-Byzantine processes.

2 Formal system model and properties

Before turning to the RandSolomon protocol description, let us first duly formalise the system model as well as a set of properties that a protocol must have to be considered a distributed Byzantine fault-tolerant random number generator.

Our system is made up of N nodes which run our protocol as a process which executes a prescribed sequence of steps. Among the participants, a portion $f < \frac{N}{3}$ of them might be Byzantine who can collaborate with each other but have limited computing power.

The nodes can communicate with each other via messages that are sent through a point to point network. This network is available for all running processes and guarantees that if a message is sent through a channel, then it must be eventually delivered (in agreement with the partial-synchrony assumption). Whenever a process executes a broadcast it does so by just sending a message to every other process (we use a best effort broadcast).

Recall that in a BFT-RNG protocol, every process proceeds through clearly demarcated phases: (1) generation and commitment, (2) reveal, and (3) result computation. A phase begins with the first correct process entering it. In this setup, a BFT-RNG protocol satisfies the following properties:

- **Agreement.** Every correct process decides on the same random number;
- **Unpredictability.** Before the beginning of the reveal phase, no process can distinguish an execution that generates $RAND$ as a random number, from an execution that generates $RAND'$, for any $RAND' \neq RAND$;
- **Randomness.** The values decided by correct processes follow a uniform distribution;
- **Termination.** Eventually, every correct process decides on a value.

Although not an intrinsic property of *BFT-RNG*s, our protocol differs from existing protocols because it provides **retraceability**. It means that after the reveal phase, a process can verify that all the steps taken to generate the shared data used to produce the final random number were correctly followed.

3 The RandSolomon protocol

Overview. From a high-level viewpoint, the protocol aggregates enough locally generated random numbers, so that enough inputs inputs are truly random and the final result observes all the properties desired. Numbers are produced locally, then encoded using an erasure correcting code and encrypted before sharing. All non-Byzantine processes agree on which numbers should be used by solving consensus, while the result remains secret (sealed under an encryption layer) as no process holds all the information necessary for computing it prior to the reveal phase. The protocol cannot be stopped by f (or less) Byzantine processes, as prior to the consensus the progress of correct processes depends solely on themselves and after it, thanks to our use of the erasure correcting code, the correct processes can retrieve data without using the information held by their Byzantine counterparts.

Notation. We shall use $[N] = \{1, 2, \cdots, N\}$, $(\cdot)_i$ to indicate that the value enclosed by the parenthesis contains a signature of process p_i and $\{\cdot\}_i$ to indicate that the value enclosed by the curly brackets was encrypted using p_i's public key. Furthermore, b will denote the number of symbols in the encoded value to be encrypted in a given encryption key; z the size of the symbols used in a code; t is the number of erasures a code can correct; l is the length of a code; d the number of data symbols in a code.

3.1 Primitives

The system requires a *deterministic* encryption infrastructure where every process knows the public key of every other processes in the system, but each of them maintains its private key secret. *Deterministic* means here that at every time two processes encrypt the same number using the same key, they get the same result [2].

Although the use of deterministic encryption is crucial for the correct execution of the protocol, these primitives are used only to encrypt long-enough (at least 256 bits) sequences of uniformly random bits. As such, the source of randomness in cleartext mitigates the security issues which crop up when using deterministic encryption [23].

We use a *consensus* protocol to ensure that each correct process disposes of the same information. The consensus protocol used here must ensure that eventually every correct process outputs a value (Termination) and that not two correct processes outputs different values (Agreement). Further, the protocol must ensure *external validity* [9]: only a *valid* value can be output, i.e., the output must satisfy a predefined *valid* predicate:

▶ **Definition 1** (Predicate valid). *valid*(v) *is true iff* v *contains* $N - f$ *inputs signed by* $N - f$ *different processes.*

Any partially-synchronous algorithm that tolerates f Byzantine failures among $3f + 1$ processes can be used [7, 32, 14].

Finally, let us consider a different perspective on secret sharing mechanisms [28]. In a classical Shamir secret-sharing protocol, when a dealer shares a secret s with N processes $p_1, p_2, \cdots, p_N$ using a threshold of $N-t$, it sends the shares $s_1, s_2, \cdots, s_N$ to their respective processes. Any $N - t$ of these shares are sufficient to retrieve s, while less than $N - t$ can reveal nothing on the secret in question. Indeed, one could consider the string $s_1 s_2 \cdots s_N$ as a code, the non-received values as erasures and hence conclude that, in fact, the secret sharing scheme can be also analysed as an Erasure Correcting Code capable of correcting t erasures [20].

In Information Theory, the number of substitutions required to change one string into another is known as *Hamming Distance* [19]. We can then conclude that we need in fact an Erasure Correcting Code with Hamming distance at least $t + 1$. The class of error-erasure correcting codes known as *Reed-Solomon (RS)*[24] with the required distance is capable of correcting t **erasures** (notice we do not treat it as an error correcting code, but an erasure correcting: an error correcting code is capable of correcting a string with corrupted data placed in unknown locations, while an erasure correcting code needs to know the positions of the string which were corrupted). Therefore, this class provides optimal block size known as *Singleton Bound* [29]. From a more pragmatic viewpoint, Reed-Solomon codes have free library implementations in many programming languages, they have deterministic parameters and encoding which are ideal for our requirements. Furthermore, most applications running our protocol will have relatively small block sizes and one can enhance the performances through hardware implementations [17]. It should be noted however, that any code complying with the following *Abstract Code* requirements can be used in our protocol.

Abstract Code.
- Have a code-word of size $b \times N$ symbols;
- Be able to correct up to $b \times f$ symbol erasures;
- $b \times z \geq 256$

Considering that we make use of Reed-Solomon codes we briefly present their general parameters:

Abstract Reed-Solomon code.
- The symbols have size z bits
- The data has length d symbols
- The code-word has length l where $l \leq 2^z - 1$ symbols
- It can correct up to t erasures where, $t = l - d$

Adjusting the above *Abstract RS code* to match the *Abstract code* and the system requirements, leads to the following Concrete Reed-Solomon Code which is suitable for implementing our protocol.

Concrete Reed-Solomon code.
- The symbols have size z bits;
- Each block to be encrypted has a size b of at least $\frac{256}{z}$ symbols;
- The data has a length of $b(N - f)$-symbols;
- The code-word has a length of $b \times N$ symbols.

It should be noted that as our protocol allows correct processes to retrace the execution followed by Byzantine processes and detect when they generate incorrect messages, we can use erasure correction instead of error correction. This drastically improves the coding performance as every error-erasure correcting code can correct two times more erasures than errors. This has two implications on our protocol: first we need fewer parity bits; second, if we were to unnecessarily use the code for errors correction, the protocol would only tolerate up to $\lfloor \frac{N-1}{4} \rfloor$ Byzantine processes. The reason for the potential loss of resilience comes from the fact that we would need to correct $2f$ errors: f errors introduced by the Byzantine member during the generation and f more for the missing blocks due to asynchrony. Therefore the number of parity blocks would have to be at least $2(2f) = 4f$ blocks, while the code must have length N blocks. Because the length of a code is larger than the number of parity symbols, $N > 4f$. This illustrates the contribution of retraceability: it implies simpler data reception by eliminating the need to generate proofs and to check them, and guarantees better resilience whilst maintaining the correctness of the protocol.

3.2 Algorithm

Generation and commitment. Each process p_i taking part in the protocol begins by generating a random number r_i of $b(N - f)$ symbols and encoding it using a Reed-Solomon encoder complying with the specification given in subsection 3.1 obtaining a number s_i of $b \times N$ symbols (lines 1, 2). This encoded number s_i is then split in N blocks of b symbols and each of these blocks are encrypted using the public key of the different processes in the system in order, signing the final result and obtaining the variable $\underline{s_i}$ (line 3).

Each process p_i share their $\underline{s_i}$ (line 4) and collect $N - f$ numbers of this type, coming from $N - f$ distinct processes according to their signatures. With this set of $N - f$-numbers they can engage in consensus and learn the same set, say RNL, of $(N - f)$ numbers generated by $N - f$ distinct processes (line 6).

Algorithm 1 RandSolomon code for process p_i.

Each function is entirely executed before executing the next
Static Local Variables:
$RNL := 0$: set of encoded and encrypted shared random numbers learnt in Consensus
$SEEN := \emptyset$: map where the key is the index of a process and the value is the value it produced
$\sigma_i[1..N][1..N] := \perp$: array of plain random number shares used in reconstruction
$RAND_i := 0$: random number decided by p_i

{Generation and Commitment Phase}
1 Generate random number r_i of $b(N-f)$ symbols of z-bits
2 Encode r_i into s_i with Desired RS
3 $\underline{s_i} = (\{s_i[1]\}_1, \{s_i[2]\}_2, \cdots, \{s_i[N]\}_N)_i$
4 Broadcast $\langle GENERATED, \underline{s_i} \rangle$

upon receiving $\langle GENERATED, \underline{s_j} \rangle$
5 $SEEN[j] := \underline{s_j}$
6 **if** $|SEEN| = N - f$ **then** $RNL := Consensus(SEEN)$

{Reveal Phase}
upon $RNL \neq \emptyset$
7 $\forall \underline{s_j} \in RNL$ **do**
8 Decrypt $\underline{s_j}[i]$ from $\underline{s_j}$ into $s_j[i]$
9 $\sigma_i[j][i] := s_j[i]$
10 Broadcast $\langle REVEAL, (\sigma_i[:][i])_i \rangle$

upon receiving $\langle REVEAL, (\sigma_j)_j \rangle$, $j \neq i$ **execute after** $RNL \neq \emptyset$
11 $\forall \underline{s_k} \in RNL$ **do**
12 **If** $\{\sigma_j[k][j]\}_j = \underline{s_k}[j]$ from $\underline{s_k}$ **then** $\sigma_i[k][j] := \sigma_j[k][j]$

{Result Computation Phase}
upon $RNL \neq \emptyset \wedge \forall \underline{s_j} \in RNL, \exists K \subseteq [N], |K| = N - f : \sigma_i[j][k] \neq \perp$
13 $step := 0$
14 $PRE := 0$
15 $\forall \underline{s_j} \in RNL$ sorted by j **do**
16 Decode $\sigma_i[j]$ into $\tilde{r}_j$ using Desired RS
17 **If** $\tilde{r}_j$ encoded with Desired RS and blockwise encrypted doesn't match $\underline{s_j}$ **then** $\tilde{r}_j := 0$
{Circular right shift by *step* blocks or $b \times step$ symbols}
18 $PRE := PRE \oplus (\tilde{r}_j \gg step)$
19 $step$++
{XOR blocks pairwise with triple in the end if necessary}
20 **for** $k := 1;\ 2k - 1 < N - f;\ k := k + 2$
21 $RAND_i[k] := PRE[2k-1] \oplus PRE[2k]$
22 **if** $2k - 1 = N - f$ **then** $RAND_i[k] := RAND_i[k] \oplus PRE[N - f]$
23 Decide $RAND_i$

Reveal. After obtaining the *RNL* set, each process can decrypt the blocks it is responsible for (line 8) and reveal them to the system via a broadcast (line 10).

(A best-effort broadcast in which a process simply sends the message to every other process will suffice.)

The processes gather the shares necessary for decoding the erasure correcting code, making sure that they truly are the decrypted versions of the RNL shares (line 12).

Result computation. Once a process has gathered at least $N - f$ shares of each of the numbers in the RNL set, it can reconstruct all of them (line 16). If the decoded version $\tilde{r}_j$ of a RNL number is again encoded and encrypted, leading to the same value for $\underline{s_j}$, then this implies that any $N - f$ shares obtained by any correct process will give the same $\tilde{r}_j$ making it consistent to be used in the final step computations.

Importance of verification. Notice that if p_i is Byzantine, then it can generate a number r_i and insert f blocks with errors in s_i. By colluding with other Byzantine processes in the system, a correct process p_j might get no response from f Byzantines and get these f erroneous blocks, essentially receiving a number with $2f$ incorrect blocks, which leads it to decode a number $\tilde{r}'_i \neq r_i$. Meanwhile a process p_k can get the Byzantine processes' correct shares instead of the blocks with errors, decoding $\tilde{r}''_i = r_i$, which would lead these two different correct processes producing two different random numbers in the end. This attack is nullified by the simple verification done in the line 17 and setting this number produced by a Byzantine process to 0, which is done by every process. It should be noted that because at least $N - f$ numbers are used and that there are at most f Byzantines, at least $f + 1$ numbers will not be nullified.

Cyclic XOR. Finally the correct processes will hold the same decoded versions of the RNL numbers which are well formed and can produce the same final random number by first cyclically shifting each number to the right by increasing steps of blocks (remember a block has b symbols) and then taking an XOR of them (line 18). Here, the reason for the shift is that for Byzantine processes might know the full contents of up to f numbers and f positions from each of the other numbers before the reveal phase. Assuming all the numbers produced by Byzantines were chosen, then the shift ensures that at least $f + 1$ different positions from the numbers created by correct processes will be used, hence including at least one unknown value for the malicious participant before the reveal.

Pairwise (triple) XOR and decision. The final step is to XOR the last three blocks together and the remaining blocks pairwise when $N - f$ is odd and XOR all the blocks pairwise when $N - f$ is even. Suppose this last step was not taken and the shifted XOR blocks were returned. Then if the $2f$ positions known by the Byzantine could potentially be used in the computation of a position *pos* in the result and these blocks XOR to a value x, they can assure that by promoting any unknown value different than $x \oplus y$ to be the last operand used in *pos* assures that the value y will not appear in *pos*. Because of the deterministic encryption they can immediately check the candidate values for being different than $x \oplus y$, although it is computationally unfeasible to determine their value. In our solution, however, because we guarantee that the Byzantine do not know at least two values used, there are $2^{b \times z}$ pairs that XOR to any given value and it is unfeasible to test the two values for being different than all of them (as $b \times z \geq 256$ in real scale instantiations of the protocol), let alone read, which would take $2^{2 \times b \times z}$ tests.

3.3 Execution example

We present now an example of a possible execution of our protocol with one Byzantine process and four processes in total illustrated in Figure 1. For pedagogical reasons we assume that the symbols have 8-bits and that each block to be encrypted contains 1 symbol ($b = 1, z = 8$), relaxing the requirement that $b \times z \geq 256$.

The beginning of the protocol and the *Generation and Commitment Phase*, corresponding to lines from lines 1 to 3 of the algorithm is shown in Figure 1a. The correct processes p_1, p_2 and p_3 produce each a 3 bytes random number, correctly encoding into a 4 bytes reed-solomon codeword. The values $\underline{s_1}, \underline{s_2}, \underline{s_3}$ ready to be shared are obtained by encrypting each of the 4 bytes from the codewords with the public keys of the p_1, p_2, p_3 and p_4, respectively. On the other hand, process p_4, who is Byzantine, maliciously produces two bad values: $\underline{s'_4}$ with an error in its third byte and $\underline{s''_4}$ with an error on its second byte.

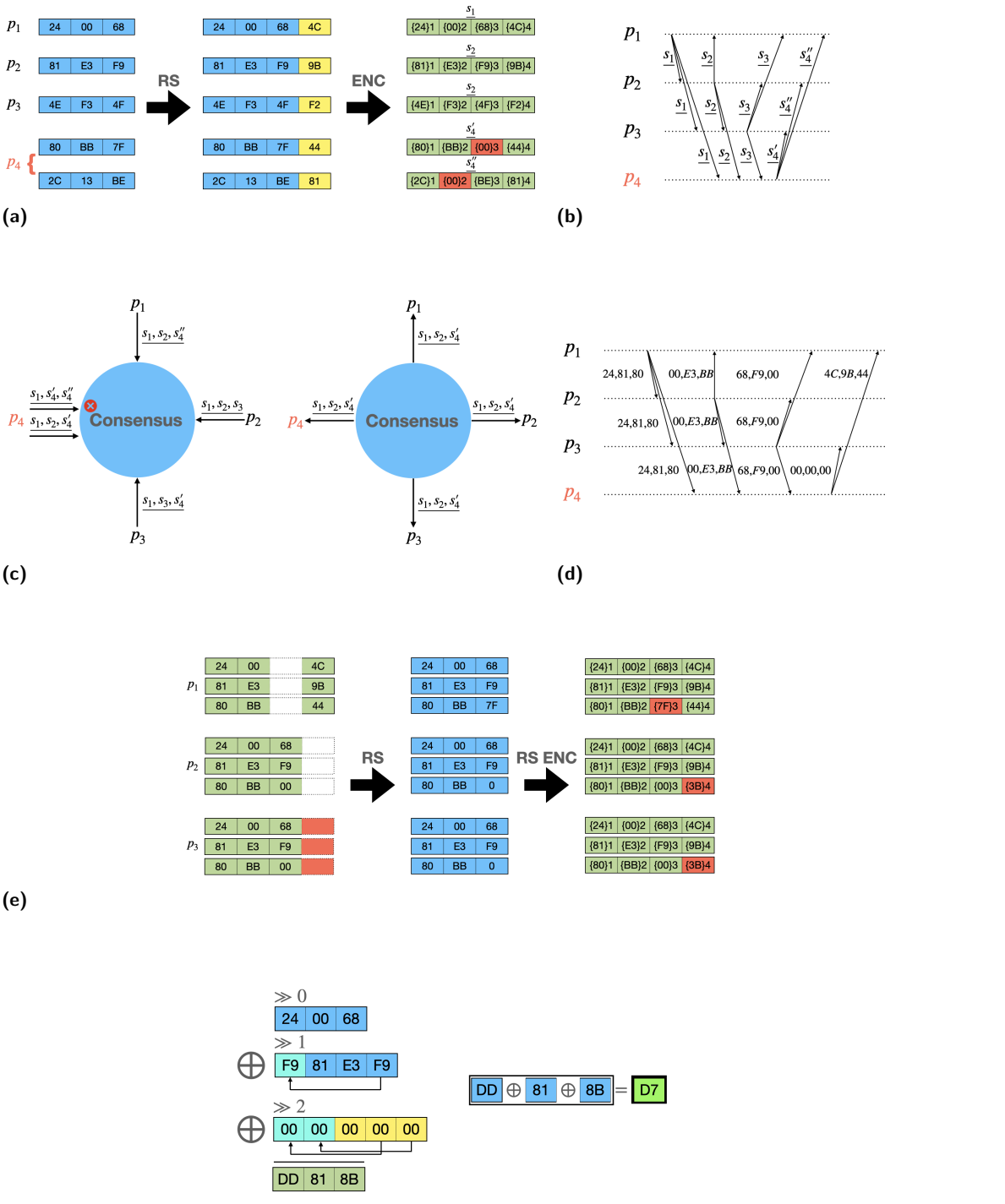
Figure 1b then shows lines 4 and 5 where processes share their produced values and collect values coming from other processes. Notice that contrary to correct processes, Byzantine processes might send different values to different destinations.

Once each process has gathered three (N-f) different values, they propose what they know to the consensus component (line 6 and Figure 1c). Nothing prevents the Byzantine process p_4 of making more than one proposal to consensus, but any proposal which is not composed by $N - f$ signatures is discarded. Once the consensus algorithm terminates, any valid value might be returned, but all processes will get the same result (decided value equal to $\underline{s_1}, \underline{s_2}$ and $\underline{s'_4}$).

The *Reveal Phase* illustrated in Figure 1d then begins, comprising lines 7 to 12. At this point processes openly share the symbols that were previously encrypted in their public keys. One deviation Byzantine processes might do is to send wrong numbers that do not correspond to the agreed values counterparts. However, because of the deterministic encryption, the receiver can detect it by asserting that the encrypted version does not match the plain value received and discard it. Moreover, even if the Byzantine process does not send its share to every participant it does not matter, as $N - f$ shares are available nonetheless.

Once processes gather three shares for each of the numbers agreed upon in consensus they can start the *Result Computation Phase* executing lines 15 to 23. Figure 1e shows how they first obtain the decoded version of the numbers and then redo both the reed-solomon encoding and the encryption of the blocks to check that they correspond to the value decided in consensus. At this point they discard the value generated by p_4 nullifying its contribution and computing the final random number by XORing the other values as shown in Figure 1f.

On the right column of the same figure we can see that the processes sort the agreed numbers by their origin, in this case they take r_1, r_2 and r''_4 in this order. They proceed by cyclically shifting the first number by 0 blocks, the second by 1 block and the last by 2 blocks. They obtain the same number $DD, 81, 8B$ and produce the same final random number $D7$ by XORing all three blocks, as these are the last three blocks. Note that if the system had $f = 3$ and $N = 10$, for example, the result from the cyclic XOR would have $N - f = 7$ blocks $B_1|B_2|B_3|B_4|B_5|B_6|B_7$ and the final random number would have three blocks: $B_1 \oplus B_2|B_3 \oplus B_4|B_5 \oplus B_6 \oplus B_7$.



(a) (b) (c) (d) (e) (f)

Figure 1 Example of a RandSolomon execution with one Byzantine process among a system of 4 processes.

4 Formal analysis of the protocol

4.1 Correctness

This section is devoted to the proof that RandSolomon is a correct partially-synchronous BFT-RNG. We do so by showing that the protocol satisfies the set of properties stated in Section 2.

▶ **Proposition 1.** *RandSolomon achieves* Agreement.

Proof. Because of the consensus using the external validity property , every correct process has the same *RNL* set. Correct processes then use shares that have been verified and match the values agreed upon (line 12), allowing them to only access the original values generated in line 2.

If a RNL number s_j passes the test in line 17, any $N - f$ correctly decrypted shares of this number shall yield the same number, as the encoded value contains no errors. It follows that every correct process will only use correctly decrypted shares and every correct process will hold the same number $\tilde{r}_j$ which will pass the test by our hypothesis.

If, however, this RNL number s_j does not pass the test, then there is an error in its encoding, as the test is merely checking if it was correctly done, and it will be visible to all correct processes in the system which will all proceed to ignore this number.

Therefore all $RAND_i$ are equal, as they are formed by XORing and shifting the same RNL numbers which every correct process agrees upon. ◀

▶ **Proposition 2.** *RandSolomon achieves* Unpredictability.

Proof. A process with limited computational power has negligible probability of determining the plain value corresponding to an encrypted value it does not possess the decryption key of. It can however test that it does not correspond to a certain value.

If Byzantine processes collude and share each others values before the different processes agree on which $N - f$ values at the end of the generation phase will compose the final result, they will know at most f full values. They will also possess f shares of each of the remaining $f + 1$ chosen values corresponding to their positions but it is impossible for them to get any more shares prior to correct processes entering the reveal phase and sending them this information. Thus, they cannot determine the value of any given position in the decided value as the shifts makes so that at least $2f + 1$ positions from the operands are needed in order to determine a position from the result and as established, the Byzantine can know at most $2f$ of them. It can still determine that the result is different than some specific value though, but as each position is then determined by the XORed with at least one other position, this possibility is then nullified as it would require the Byzantine processes to test $2^{b \times z}$ pairs of numbers in order to eliminate a value, which is computationally unfeasible with real scale protocol parameters ($b \times z \geq 256$). ◀

▶ **Proposition 3.** *RandSolomon achieves* Randomness.

Proof. By hypothesis, correct processes are capable of generating uniformly random numbers. The result of XORing a uniformly distributed random variable X in D with a constant c in D is a uniformly distributed random variable in D. Also, the result of XORing two independent uniformly distributed variables X and Y over D is uniformly distributed. As we already established in the final two paragraphs of subsection 3.2, each position in the final result is independent from each other and uses at least two uniform random numbers coming

from correct processes unknown to the Byzantine before the reveal phase. This means no proposed values are preferred over others and the randomness of the operands is transferred to the output. ◀

▶ **Proposition 4.** *RandSolomon achieves* Termination.

Proof. Every correct process generates their random numbers and propose a set of $N - f$ of them to the consensus component. This means that there will be at least $N - f$ processes engaging in it, and because it can tolerate up to f failures, it will eventually give all correct processes their RNL sets.

Once $N - f$ correct processes learn what the RNL set is, they will share their shards, meaning that each correct process is guaranteed to receive at least $N - f$ correct shares of each of their RNL numbers, satisfying the conditions for entering the computation phase, where their progress becomes purely local as they do not depend on other processes anymore. ◀

4.2 Complexity

We shall analyse our algorithm in terms of *message complexity*: the maximum number of messages transmitted per random number generated; *bit complexity*: the maximum number of bits exchanged over the network per random bit generated; *time complexity*: the number of message round trips required per random number generated; and *computational complexity*: the number of operations to be executed per process per random number generated.

In the generation and commitment phase, each process executes one broadcast, meaning that there are $O(N^2)$ messages being sent at this phase. After consensus is reached on the value of *RNL*, each process executes exactly one more broadcast, leaving the message complexity of this part of the protocol on $O(N^2)$. The result computation phase in done locally. Hence the message complexity of our protocol is $O(N^2)$ outside consensus.

In terms of bit complexity, RandSolomon produces random numbers of $O(N)$ bits, therefore we consider the number of bits exchanged divided by N. The messages of the generation phase contain random numbers whose lengths are proportional to the number of processes in the system by design. Therefore, the bit complexity of this step is $O(N^2)$. Afterwards in the reveal phase, each process includes one decrypted block per number in the *RNL* set. Each decrypted block has constant size and the cardinality of *RNL* is $f+1$, so the bit complexity of this stage is also $O(N^2)$. Therefore, without taking consensus into account, the bit complexity of our protocol is $O(N^2)$. The inputs for consensus are comprised of $N - f$ values of $O(N)$ bits and therefore the bit complexity (used in the Table 1) is $O(N) \times Consensus$.

With respect to time complexity, our protocol requires outside consensus two message delays given the two aforementioned broadcasts, each executed by all processes in parallel. Consensus might require *view-changes* in the worst case bringing its time complexity to $O(f)$ which corresponds to our overall time complexity. As for computational complexity we present the analysis split on the three phases of the protocol in Table 2.

Table 2 Computational complexity.

Operation	Generation	Reveal	Result
Encryption	$O(N)$	$O(N^2)$	$O(N^2)$
Decryption	0	$O(N)$	0
ECC encoding	$O(1)$	0	$O(N)$
ECC decoding	0	0	$O(N)$

If the erasure correcting code used is indeed Reed-Solomon, then the encoding and decoding complexities of a single number with length $O(N)$ is $O(N \log N)$ [30], meaning that the per-process computational complexity is $O(N^2 \log N)$ when this particular code is used.

When considering the complexity of the Consensus protocol, one can easily adopt last generation PBFT consensus protocols developed in the context of blockchain-type ledgers. In this context, the Tendermint (analysed in detail in [1]) or Hotstuff [32] consensus protocols can be used within RandSolomon. Doing so leads to an overall message complexity of $O(N^2)$ and bit complexity of $O(N^3)$ accounting view-changes with the complexity of consensus dominating that of our protocol for both protocols considered. As such, any system which already has the protocol machinery to solve consensus can implement RandSolomon without incurring a significant performance impact.

In a run where the system is synchronous (after passed GST) and the consensus leader is correct, the protocol terminates in constant number of message delays and incurs only $O(N^2)$ bit complexity, comparable to that of synchronous protocols.

The complexity analysis of RandSolomon is summarised in Table 3.

Table 3 RandSolomon Protocol complexities integrating Consensus as in [32].

Complexity	Generation	Consensus	Reveal	Result	Total
Message	$O(N^2)$	$O(N^2)$	$O(N^2)$	0	$O(N^2)$
Bit	$O(N^2)$	$O(N^3)$	$O(N^2)$	0	$O(N^3)$
Time	1 msg delay	$O(f)$	1 msg delay	0	$O(f)$
Computation	$O(N)$	$O(N)$	$O(N^2)$	$O(N^2 \log N)$	$O(N^2 \log N)$

5 Conclusion

We presented RandSolomon, a Byzantine fault-tolerant protocol capable of generating a common random number in a partially-synchronous system. As we have previously shown in section 1, although the problem of generating randomness in multi-party systems has already been extensively discussed, the partially-synchronous systems still lacked a BFT solution with the optimal resilience of f Byzantine participants among $3f + 1$ with deterministic termination. Not only did we provide such a solution but we also employed very simple public key cryptography, not relying on a random oracle, by means of what we have called *retraceability*. Our approach is modular, using Consensus as a black box, which facilitates future implementations of the protocol with improved complexity metrics.

References

1 Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Dissecting tendermint. In *International Conference on Networked Systems*, pages 166–182. Springer, 2019.

2 Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In *Annual International Cryptology Conference*, pages 535–552. Springer, 2007.

3 Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 390–399, 2006.

4 Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Annual international cryptology conference*, pages 757–788. Springer, 2018.

5 Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

6 Ilja N Bronshtein and Konstantin A Semendyayev. *Handbook of mathematics*. Springer Science & Business Media, 2013.

7 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint*, 2018. `arXiv:1807.04938`.

8 Benedikt Bünz, Steven Goldfeder, and Joseph Bonneau. Proofs-of-delay and randomness beacons in ethereum. *IEEE Security and Privacy on the blockchain (IEEE S&B)*, 2017.

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. Cryptology ePrint Archive, Report 2000/034, 2000. URL: `https://eprint.iacr.org/2000/034`.

11 Ignacio Cascudo and Bernardo David. Scrape: Scalable randomness attested by public entities. Cryptology ePrint Archive, Report 2017/216, 2017. URL: `https://eprint.iacr.org/2017/216`.

12 Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 66–98. Springer, 2018.

13 Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.

14 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3132747.3132757`.

15 Andreas Haeberlen and Petr Kuznetsov. The fault detection problem. In *International Conference On Principles Of Distributed Systems*, pages 99–114. Springer, 2009.

16 Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018. `arXiv:1805.04548`.

17 MA Khan, S Afzal, and R Manzoor. Hardware implementation of shortened (48, 38) reed solomon forward error correcting code. In *7th International Multi Topic Conference, 2003. INMIC 2003.*, pages 90–95. IEEE, 2003.

18 Mikhail Krasnoselskii, Grigorii Melnikov, and Yury Yanovich. No-dealer: Byzantine fault-tolerant random number generator. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 568–573. IEEE, 2020.

19 Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error correcting codes*, volume 16. Elsevier, 1977.

20 Robert J. McEliece and Dilip V. Sarwate. On sharing secrets and Reed-Solomon codes. *Communications of the ACM*, 24(9):583–584, 1981.

21 Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*, pages 120–130. IEEE, 1999.

22 Thanh Nguyen-Van, Tuan Nguyen-Anh, Tien-Dat Le, Minh-Phuoc Nguyen-Ho, Tuong Nguyen-Van, Nhat-Quang Le, and Khuong Nguyen-An. Scalable distributed random number generation based on homomorphic encryption. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 572–579. IEEE, 2019.

23 Charles Rackoff and Daniel R Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Annual International Cryptology Conference*, pages 433–444. Springer, 1991.

24 Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

25 Arto Salomaa. *Public-key cryptography*. Springer Science & Business Media, 2013.

26 Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. Hydrand: Practical continuous distributed randomness. Cryptology ePrint Archive, Report 2018/319, 2018. URL: https://eprint.iacr.org/2018/319.

27 Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Annual International Cryptology Conference*, pages 148–164. Springer, 1999.

28 Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

29 Richard Singleton. Maximum distance q-nary codes. *IEEE Transactions on Information Theory*, 10(2):116–118, 1964.

30 Alexandre Soro and Jérôme Lacan. FNT-based Reed-Solomon erasure codes. In *2010 7th IEEE Consumer Communications and Networking Conference*, pages 1–5. IEEE, 2010.

31 Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 444–460. Ieee, 2017.

32 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.