Self-Testing/Correcting Protocols

(Extended Abstract)

MATTHEW FRANKLIN[⋆] JUAN A. GARAY[⋆⋆] MOTI YUNG[⋆⋆⋆]

Abstract. In this paper we suggest the notion of *self-testing/correcting protocols*. The work initiates the merge of distributed computing and the area of "program checking" introduced by Blum, and specifically employs extended notions from the work of Blum, Luby and Rubinfeld. In this setting, given a protocol P (a collection of programs on a network of n processors) which allegedly implements a distributed function f, a *self-tester* for f is a (simpler) protocol which makes calls to P to estimate the probability that P when executed in a given environment is faulty (i.e., P and f differ in some of the outputs). A *self-correcting* protocol is another protocol which allows for the computation of f correctly on every input (with high probability) as long as P in the same type of environment is not too faulty.

We first consider self-testing/correcting under a basic form of environmental malfunction, that of *crash* failures, and design a self-tester/corrector pair for protocols implementing the agreement "function." Many distributed protocols can be designed "on top" of this primitive, and can be self tested/corrected whenever it can be. We then consider self-testing/correcting under *gossiping* failures, and present a generic self-testing/correcting pair that is *privacy-preserving*. The notion is basic in protocols where secrecy is an issue. A self-corrector for P is privacy-preserving if it is private (with overwhelming probability) whenever P is private (with overwhelming probability).

In the process of our study, we identify the basic components of a protocol self-testing "utility library," which allows for the safe bootstrapping of the self-testing/correcting process.

1 Introduction

Traditional program testing involves running the program on a "random" subset of the possible inputs, for which the correct outputs are already known. There are several drawbacks to this approach. First, it is not clear what "random" means in this context (equally-likely inputs?; a distribution that arises in practice?; is this distribution well defined or known at all?; etc.). Both Kannan [20] and

⋆ Xerox PARC, 3333 Coyote Hill Rd., Palo Alto, CA 94304. E-mail: `franklin@parc.xerox.com`.

⋆⋆ Bell Labs, 600 Mountain Ave, Murray Hill, NJ 07974. E-mail: `garay@research.bell-labs.com`. URL: `www.bell-labs.com/user/garay/`.

⋆⋆⋆ CerCo Inc., 55 Broad St., New York, NY 10004. E-mail: `moti@cs.columbia.edu`.

Lipton [23] point to the extreme case of a program that is wrong on exactly one input. Such a program is extremely likely to pass the testing stage, and thus when the program is run on the bad instance the program's output is accepted as correct. The second problem concerns the requirement that some of the outputs be known. How is this achieved? In some cases instances are generated that are "small" enough that can be checked by hand and this is insufficient to make an assertion about the general behavior of the program. In other cases a "different" program is used for the same computation. How independent is this program from the one being tested? How was the confidence about the correctness of this program itself obtained?

Attempting to cover some of the above criticism, Blum [4] proposed a new framework called *program result checking* (program checking, for short). Program checking involves writing a result checker program to be run in conjunction with the original program, that checks the work of the program. Program checking is instance specific: each time the program is run on an input, the checker gives us confidence that the program is correct on *that* input. I.e., the program checker does not verify whether the program is correct or buggy as a whole; it verifies whether the program gives the correct answer on the particular inputs on which it is invoked. Since checking is done during each run of the program, it remedies the program verification problem in that the actual running version—i.e., the compiled version and the hardware on which it runs—of the program is evaluated.

Building on program checking, Blum, Luby and Rubinfeld [11] introduced the notion of *self-testing/correcting programs*. A *self-tester* is a (oracle) program that makes calls to the program P to be evaluated to estimate the probability that $f(x) = P(x)$ for random x. Although a program checker can be used to verify whether P is correct on particular inputs, it does not provide a method for computing the correct answer when the program is faulty. A *self-corrector* corrects P as long as P's probability of error, estimated by the self-tester, is sufficiently low. Blum, Luby and Rubinfeld develop general techniques for constructing simple to program self-testing/correcting pairs for a variety of numerical problems [11].

Independently, Lipton [23] also discussed the concept of self-correcting programs and used it to derive a distribution-free theory of random testing; for several problems he constructed testing programs with respect to any distribution assuming that the programs are not too faulty with respect to a particular distribution. Instrumental in both developments above is the *random self-reducibility* property. Informally, a function f is random self-reducible if, for any x, the computation of $f(x)$ can be reduced to the computation of f on other "randomly chosen" inputs.

Protocol self-testing. In this paper we extend the Blum methodology to deal with *distributed protocols*. Such protocols should be understood here as collections of programs running in a network of n processors, one for each processor. Motivated by [11], we introduce the notion of *self-testing/correcting protocols*. Given a protocol P which allegedly implements a distributed function f, a *self-tester* for f is a protocol which makes calls to P to estimate the probability that P is faulty (i.e., P and f differ in some of the outputs). A *self-correcting* protocol is another protocol which allows for the computation of f correctly on every input (with high probability) as long as P is not too faulty.

In such an environment, other things can go wrong besides the faulty behavior due to flaws in the design and coding of the protocol (e.g., processor failures, messages delays, messages being lost, etc.), making the task of evaluating the correctness of a protocol more complicated than in the case of a sequential program. In particular, the benefit of evaluating the running version of a protocol and the "generalized hardware" on which it runs becomes all the more relevant in this context.

As in [11], we require that the self-tester be *different* than any protocol implementing the function under consideration, as well as *simple*, and *efficient* (see Section 2). The first requirement guarantees the independence of the verification step, the second (however difficult to quantify) makes it easier to assess its correctness, while the third assures that the cost for using it is not too high (e.g., big slowdown in the running time) so as to overwhelm the benefits of using it.

We first consider (Section 3) self-testing/correcting under the basic form of environmental malfunction of crash failures, and design a self-tester/corrector pair for the agreement "function" (aka, Byzantine agreement, Distributed Consensus [24]).

In [11, 25], Blum *et al.* discuss the notion of using several programs—a *library*, all of which are possibly faulty, to aid in the testing and correcting. In the process of designing our self-tester/corrector pair for agreement, we identify some of the basic components (co-routines) that a protocol self-testing "utility" library should contain; besides agreement, these include a shared random coin (e.g., [13]) and Crusader agreement [14]. Such a protocol library, together with [11]'s program library, enable the **bootstrapping** of self-testing/correcting for other protocols.

One important class of protocols are those which enable secure multi-party computation [9, 12]. Besides the correctness requirement, such protocols are required to be *t-private*; informally, this means that an adversary viewing the transcripts of any t parties learns nothing about the computation beyond what can be inferred from their inputs together with the result. From a security standpoint, it is important that self-testers and self-correctors do not create new vulnerabilities. We also present in this paper (Section 4) a generic self-testing/correcting pair that is *privacy-preserving*: if a protocol is t-private, then so is its self-testing/correcting pair.

Related work. To the best of our knowledge, this is the first time that the self-testing/correcting of protocols is discussed.

Program checking and self-testing have been extensively investigated, and checkers and self-testers are now known for many problems [10, 2, 19, 26, 27, 17]. Other relevant works include checking the correctness of memory [6], cryptographic program checking [18], and batch checkers [8].

In the related but different area of *self-stabilizing protocols* (e.g., [1, 3, 15, 22]) the notion of "checking" has been considered as an important co-routine.

This setting assumes a "correct protocol control" constantly checking the state of the execution, which stabilizes when faults cease to exist. The self-stabilizing approach as a whole does not propose a "tool" for reliability; rather, it is a "protocol design methodology" for certain faulty environments. Our setting is quite different: we apply our approach as a reliability tool which checks and actively corrects outputs in the presence of faulty control and executional environment.

2 Model and Definitions

The model we consider in this paper is a synchronous, fully connected network of n processors/parties with identities from $\{1, 2, \cdots, n\}$.

The literature on checking calls the problem specification a "function" while the implementation is called a "program." In a distributed environment, a function may admit more than one outcome depending on the environment's "input." This is a behavior which can be specified using relations; however, for consistency with the existing theory of sequential function self-testing/correcting, we continue to model this process using functions, as explained in the following.

Distributed functions and protocols. To model the computation we define an *ideal* distributed function $\phi : V^n \to R^n$, for arbitrary domain V and range R; typically, $V = R$. A prime example of such a function is the addition function where the inputs are values from a given group and the result is the value of the sum of the inputs. Many arithmetic functions like addition have been treated in the sequential case of self-testing/correcting programs.

In a distributed environment, the behavior of individual processors and the communication environment may deviate from an ideal execution. This gives rise to specifying problems under certain assumptions on (partial properties of) the "behavior." Many times we specify that depending on this behavior the computation will take into account only a partial portion of the inputs, or we specify that only the results which are computed by the non-faulty processors be considered. For example, a "distributed addition" in a faulty environment will specify that we sum up only the values of processors that are non-faulty (up to a certain time), and that the summation result will be written by all the non-faulty processors.

We model this process through the notion of an *environment-aware* distributed function $f : W^n \to S^n$. where f corresponds to ϕ, and they agree on ideal environments. W is an extended set of inputs which includes the actual input set V, but also relevant environmental characteristics that tag each processor; these characteristics influence the outcome of the function. S extends R in a similar fashion. A protocol for a distributed function ϕ is a distributed program which implements f.

We use $\mathcal{E}$ to denote the set of *environments* where the protocols run. Examples of sets of environments are adversarial settings where processors may fail; settings where messages may be lost or delayed; etc. We use the expression "running in $\mathcal{E}$" as a shorthand for "running in environments from $\mathcal{E}$." $\mathcal{E}$ implies the relevant environmental characteristics in the specification of the extended input/output pair W/S, though $\mathcal{E}$ may have additional elements and events. For example, we may only be interested in the case of when a processor has failed (e.g., before or after a certain round) in order to characterize the set of relevant inputs; we may be interested only in whether a processor has failed or not in order to specify a condition on its output; we may not be interested in what a failed processor has written in its internal state; etc.

Oracle protocols. We will now discuss a notion of protocol extension where one protocol employs another protocol as an oracle (I/O access only). This is analogous to the formalization of program extension in [11].

Definition 1 (Oracle protocol). *A protocol $\mathcal{P}$ is an* oracle protocol *if it makes calls to another protocol P that is specified at run time. When a call is made to P, each party learns only its own output, and not any intermediate messages or computations of P (i.e., the call is "black-box"). We let $\mathcal{P}^P$ denote protocol $\mathcal{P}$ making calls to protocol P.*

We let D_V denote the probability distribution according to which the inputs v_i, $1 \leq i \leq n$, are drawn from V. If P is a protocol, then we denote $P[i]$ the output of P on processor i. Let $\texttt{error}(f, P, D_V, \mathcal{E})$ denote the probability that $f \neq P$ (i.e., $f[i] \neq P[i]$ for at least some i) on arguments randomly chosen according to probability distribution D_V running in $\mathcal{E}$. Let β be a confidence parameter.

Definition 2 (Self-testing protocol). *Let $0 \leq \epsilon_1 < \epsilon_2 \leq 1$. An (ϵ_1, ϵ_2)-*self-testing protocol *for f with respect to D_V in $\mathcal{E}$ is a probabilistic oracle protocol $\mathcal{T}_f$ that for any P running in $\mathcal{E}$ and β:*

1. *If $\texttt{error}(f, P, D_V, \mathcal{E}) \leq \epsilon_1$ then $\mathbf{Pr}(\forall i : \mathcal{T}_f^P[i] =$ "PASS"$) \geq 1 - \beta$;*
2. *if $\texttt{error}(f, P, D_V, \mathcal{E}) \geq \epsilon_2$ then $\mathbf{Pr}(\forall i : \mathcal{T}_f^P[i] =$ "FAIL"$) \geq 1 - \beta$.*

Definition 3 (Self-correcting protocol). *Let $0 \leq \epsilon < 1$. An ϵ-*self-correcting protocol *for f with respect to D_V in $\mathcal{E}$ is a probabilistic oracle protocol $\mathcal{C}_f$ that for any P running in $\mathcal{E}$ and β:*

If $\texttt{error}(f, P, D_V, \mathcal{E}) \leq \epsilon$ then $\mathbf{Pr}(\forall i : \mathcal{C}_f^P[i] = f[i]) \geq 1 - \beta$.

A *self-testing/correcting pair* for f in $\mathcal{E}$ is a pair of probabilistic oracle protocols $(\mathcal{T}_f, \mathcal{C}_f)$ such that there are constants $0 \leq \epsilon_1 \leq \epsilon_2 \leq \epsilon < 1$ such that $\mathcal{T}_f$ is an (ϵ_1, ϵ_2)-self-testing protocol with respect to D_V, and $\mathcal{C}_f$ is an ϵ-self-correcting protocol with respect to D_V.

As in [11], $\epsilon_1 = 0$ is sufficient for assuming that f will be computed correctly. On the other hand, ϵ_2 should be as close to ϵ_1 as possible in order to allow protocols that are as "buggy" as possible, and still have the corrector work properly.

Modes of operation. In order to allow for self-testing/correcting, the oracle protocol is invoked a number of times. We will consider two modes in which the oracle protocols run:

- **The "reset" mode:** The oracle calls are made sequentially and the system is *reset* with every call. Namely, the system starts from an initial state which is presented to the oracle. To allow the processors to make conclusions based on multiple oracle calls, we will assume that each processor has non-volatile memory preserving information across calls.
- **The "parallel" mode:** The oracle calls are made in parallel. Namely, the processors may have a pre-processing stage after which they invoke the oracle, running in parallel a number of executions.

Complexity—the "little oh" constraint. As in [11], we require that the self-tester/corrector be quantifiably *different* from any correct protocol that implements f, in the sense that their complexity, not counting the cost in the calls to P, should be lower than the complexity of any correct protocol for f. In this paper we consider synchronous networks, where the computation proceeds as a series of *rounds.* Thus, we require that the running time (number of rounds) of $\mathcal{T}_f^P$ (resp., $\mathcal{C}_f^P$) be $o(R)$, where R is the minimum worst-case running time of protocols that compute f, when calls to P are counted as one round. In fact, the testers and correctors we present in this paper satisfy a stricter requirement, in that their running time is within a small multiplicative constant of the running time of P, when including the running time of P.

A similar notion can be applied to communication (e.g., number of messages).

Property-preserving oracle protocols: When a protocol runs in $\mathcal{E}$ it may satisfy certain properties. For example, it may tolerate a given number of failures of a certain type, or it may be "private" with respect to an adversary who controls the history of the execution and the view of the failing (gossiping) processors.

We would like to claim that a self-testing/correcting protocol, which employs the protocol being tested/corrected as an oracle, satisfies the same property. This notion is captured by the following definition:

Definition 4. *A self-tester $\mathcal{T}_f$ (resp., self-corrector $\mathcal{C}_f$) for P is p-*preserving *if $\mathcal{T}_f^P$ (resp., $\mathcal{C}_f^P$) satisfies property p (with overwhelming probability) whenever P satisfies property r (with overwhelming probability).*

3 Self-Testing/Correcting under Environmental Malfunction

In this section we consider environments where some of the processors in the network, up to t, may fail by crashing. The semantics of crash failures is the usual one: when a processor stops sending messages in one round, a subset of the processors receive the message. Initially, we assume that the failing processors are chosen uniformly at random among all subsets of up to t processors. This is the only random choice assumption that we make. We will denote these environments by $\mathcal{C}$. Later on we consider the standard crash failure model.

3.1 The agreement function

We now consider the design of self-tester/corrector pairs for protocols implementing the *agreement* function (BA) [24]. Recall that in this problem a distinguished source s has an initial value $v_s \in V$. It is required that upon termination, all the correct processors output the same value, and if the source is non-faulty these output values coincide with the source's input value, v_s. The problem can be defined as a function as follows: Let V denote the set of input and output values, let $B = \{\text{faulty, nonfaulty}\}$ be the set of possible processor behavior patterns, and let $T = \{\text{true, false}\}$ be the set of possible termination conditions. Then the agreement function $BA\ : (V \times B)^n \rightarrow (V \times T)^n$ such that:

$$\forall i \text{ s.t. } b_i = \text{nonfaulty}, (d_i, t_i) = \begin{cases} (v_s, \text{true}) \text{ if } b_s = \text{nonfaulty}; \\ (v, \text{true}) \quad \text{where } v \in V, \text{ otherwise.} \end{cases}$$

The first condition is usually called the Validity condition, while the second is called the Agreement condition. The behavior components are given as an example; a more refined notion of behavior may specify in which round a processor fails, when it fails which subset of remaining processors receives its message, etc.

We will deal with protocols that implement the agreement function, and their correctness. It is known that every BA protocol in the presence of crash failures requires $\geq t+1$ rounds of communication in its worst-case run (e.g., [16]). Even in benign adversarial models, like the one considered in this section, this worst-case scenario may happen with a very small probability.

In order to satisfy the "little-oh" constraint, in [11] self-testing/correcting programs are designed with code that is simpler to implement—and verify—than the function being self-tested/corrected. In our case, the simple one-round *Crusader agreement* protocol [14] will play such a role. Recall that in this problem, the sender also has an initial value which it wishes to broadcast to the remaining processors. The goal is a protocol that guarantees the same Validity condition as above, but the Agreement condidion requires that no two nonfaulty processors decide on different values unless at least one of them discovers that the sender is faulty. The definition of the CA function is similar to the one above, and we omit it for brevity.

To implement the above function obviously all the sender has to do is to broadcast its input, and hence every processor can either decide on the sender's value or detect that the sender has crashed, in which case it is allowed to decide on a default value. That is, the problem is solvable in just one round.

3.2 Self-tester for agreement: reset mode

Consider the simple oracle protocol depicted in Figure 1 for assessing *one run* of a BA protocol P. (We assume that P uses a default decision value $\perp \in V$.)

The tester consists of an initial CA call, followed by a call to P on a random input drawn according to D_V, followed by another CA. In the first CA the sender distributes its randomly-chosen input. If the sender does not crash in this round, then all the non-faulty processors learn the sender's value; on the other hand, if the sender crashes, every processor knows either the input or the fact that the sender has crashed, and hence the output of the oracle call must be the default value. In the second CA, every processor i (including the sender) distributes $P[i]$, the outcome obtained by running protocol P, to all other processors. Note that any processor participating in the second CA was not faulty during the call to P. Based on its local view of input and outputs, every processor judges the protocol by deciding on either "fail" or "not fail." It is easy to check that the tests performed cover all the possibilities of protocol malfunction.

Protocol Simple_Agreement_Tester(P, s)

s chooses $v \in_{D_V} V$
$\mathit{input} \leftarrow CA(s, v)$
$P(s, v)$
for $i = 1, \cdots, n$
  $\mathit{output}_i \leftarrow CA(i, P[i])$
let $G = \{i : \text{message received in } CA(i, P[i])\}$
if message received in $CA(s, P[s])$
  $\mathit{sender\_alive} \leftarrow \textbf{true}$
$\mathit{fail} \leftarrow \quad (\exists\, j, k \in G : \mathit{output}_j \neq \mathit{output}_k) \vee$
  $(\exists\, j \in G : (\mathit{output}_j \neq \mathit{input} \wedge \mathit{sender\_alive}) \vee$
  $(\mathit{output}_j \neq \mathit{input} \wedge \mathit{output}_j \neq \perp))$

Fig. 1. Simple agreement tester for assessing one run; code for proc. i

Assume that the number of failures that may occur in $\mathcal{C}$ is at most $t < n/2$. We can state the following about the tester of Fig. 1:

Claim 1 *If the run of P is corect, then* "not fail" *holds in all nonfaulty processors.*

Claim 2 *If the run of P is incorrect, then* "fail" *holds in every nonfaulty processor with probability $\delta > 1/2$.*

In the worst case, the misbehavior of P is witnessed by just one processor, and its probability of failing—and thus not communicating its findings to any other processor—is less than $1/2$.

We will now use the simple tester of Fig. 1 as the basis for a generic self-tester for BA. The self-tester is shown in Figure 2; it is presented in a "generic"

```
Protocol Agreement_Tester(P, ε, β)

    N ← (32/ε) ln(2n/β)
    S ← 0
    for j = 1, ···, N
    {   choose sender s_j
        Simple_Agreement_Tester(P, s_j)
        if fail
            S ← S + 1
    }
    if S/N > ε/4
        "FAIL"
    else "PASS"
```

Fig. 2. Generic self-tester for agreement; code for proc. i

form (ϵ as an input parameter). The protocol basically runs the simple tester an adequate number of times in order to guarantee similar conclusions on the nonfaulty processors. In the reset mode, processors that failed are rebooted each time. The way the sender s_j is chosen in each run is left unspecified. In environments from $\mathcal{C}$, the same sender can be used for all calls; however, the ability to choose senders in different ways will allow us to cope with other types of network adversaries (e.g., asymmetric).

Now to the analysis. Let $Y_l[i]$, $1 \leq l \leq N$, be the 0/1 random variable indicating the event *fail* = **true** in the lth call on processor i. From Claim 2,

$$\mu = E[Y_l[i]] = \delta \cdot \mathtt{error}(BA, P, D_V, \mathcal{C}) \ .$$

The following lemma quantifies the number of runs after which the assessments of *all* the processors are sufficiently accurate.

Lemma 1. *Let* $\gamma \leq 2$ *and* $N \geq \frac{1}{\mu} \cdot \frac{4\ln(2n/\beta)}{\gamma^2}$. *Then*

$$\mathbf{Pr}(\forall i : (1-\gamma)\mu \leq Y[i] \leq (1+\gamma)\mu) \geq 1 - \beta \ ,$$

where $Y[i] = \sum_{l=1}^{N} \frac{Y_l[i]}{N}$.

The proof of the lemma follows from a slight variation of the "Zero-One Estimator Theorem" of [21] for identically-distributed 0/1-valued random variables with mean μ, and the linearity of expectation. Similarly to [25], we will make use of the following corollary:

Corollary 1. (1) *Let* $\mu' \leq \mu_i$, *for all* i, *and let* $N = \frac{1}{\mu'} \cdot 16\ln(2n/\beta)$. *Then* $\mathbf{Pr}(\exists i : Y[i] \leq \mu'/2) \leq \beta$. *(Use* $\gamma = 1/2$.*)*

(2) *Let $\mu'' \geq \mu_i$, for all i, and let $N = \frac{1}{\mu''} \cdot 4\ln(2n/\beta)$. Then* $\mathbf{Pr}(\exists i : Y[i] \geq 2\mu'') \leq \beta$. *(Use $\gamma = 1$.)*

Theorem 1. *The protocol of Figure 2 is an $(\frac{\epsilon}{8}, \epsilon)$-self-tester for BA.*

Proof (Sketch). The "little-oh" requirement follows from the fact that CA requires $O(1)$ rounds. Regarding correctness, first note that, from Lemma 2, the expected output on processor i is $E[Y[i]] = \mu_i = \delta \cdot \mathtt{error}$, $\frac{1}{2} < \delta \leq 1$. There are two cases to consider:

($\mathtt{error} \geq \epsilon$) Let $\mu' = \frac{\epsilon}{2}$ (taking the lower bound on δ) and $N = \frac{1}{\epsilon} \cdot 32\ln(2n/\beta)$. Corollary 1(1) yields $\mathbf{Pr}(\exists i : Y[i] \leq \frac{\epsilon}{4}) \leq \beta$. On the other hand, in the protocol of Fig. 2 processor i outputs "FAIL" if $Y[i] > \frac{\epsilon}{4}$. Hence, if $\mathtt{error} \geq \epsilon$, the protocol outputs "FAIL" on every processor with probability at least $1 - \beta$.

($\mathtt{error} \leq \frac{\epsilon}{8}$) Let $\mu'' = \frac{\epsilon}{8}$ (corresponding to $\delta = 1$) and $N = \frac{1}{\epsilon} \cdot 32\ln(2n/\beta)$. Corollary 1(2) yields $\mathbf{Pr}(\exists i : Y[i] \geq \frac{\epsilon}{4}) \leq \beta$. On the other hand, if $Y[i] < \frac{\epsilon}{4}$ then processor i outputs "PASS." Hence, if $\mathtt{error} \leq \frac{\epsilon}{8}$, the protocol outputs "PASS" on every processor with probability at least $1 - \beta$. □

3.3 Self-corrector for agreement: reset mode

In this section we present a self-correcting protocol that given an agrement protocol P whose error probability (as estimated by the self-tester) is sufficiently low, outputs the correct BA function for any input $x \in V$ by making calls to P. For the design we follow the approach of [23]: to self-correct P on input x, the source draws at random a value r from V, processors run $P(r)$ and $P(x -_V r)$, and output $P(r) +_V P(x -_V r)$. (This is a simple case of the random self-reducibility property.) Recall from Section 3.2 that the BA function admits two outputs depending on whether the sender is faulty or not. Thus, it is necessary to guarantee that either good outcome overwhelms the erroneous output.

Assume that we are given a BA protocol P such that $\mathtt{error}(BA, P, D_V, \mathcal{C}) \leq \epsilon$ (e.g., $\epsilon = 0.05$). The ϵ-self-corrector for BA is shown in Figure 3. The protocol performs the two calls to P as described above, and then does a case analysis on the possible outcomes; this is repeated an adequate number of times.

We will need the following statement in order to establish the correctness of the self-corrector (we instantiate for reasonable values of α).

Proposition 1. *Let $\alpha \geq \frac{3}{4}$, and let $X_1, X_2, \ldots, X_N$ be independent identically-distributed 0/1-valued random variables such that* $\mathbf{Pr}[X_i = 1] \geq \alpha$, $i = 1, 2, \ldots, N$. *Then*

$$\mathbf{Pr}\left[\sum_{i=1}^{N} X_i > \frac{2N}{3}\right] \geq 1 - e^{-\frac{\alpha N}{324}} .$$

The proof follows directly from Chernoff bounds. The "2/3" captures the fact that there are two correct outcomes of the protocol that are to overwhelm the erroneous one.

Protocol Agreement_Corrector$(P, s, x, \epsilon, \beta)$

$N \leftarrow O(\frac{\ln(n/\beta)}{\epsilon})$
for $j = 1, \cdots, N$
{ s chooses $r \in_{D_V} V$
  $output_1 \leftarrow P(r)$
  $output_2 \leftarrow P(x -_V r)$
  if $(output_1 = \perp \wedge output_2 = \perp)$
    $outcome_j \leftarrow \perp$
  elseif $(output_1 \neq \perp \wedge output_2 \neq \perp)$
    $outcome_j \leftarrow output_1 +_V output_2$
  else $outcome_j \leftarrow$ '$*$'
}
$P[i] \leftarrow$ most common $outcome_j$, $1 \leq j \leq N$,
  s.t. $outcome_j \neq$ '$*$'

Fig. 3. Self-corrector for Byzantine agreement; code for processor i

Theorem 2. *The protocol of Figure 3 is an ϵ-self-correcting protocol for BA, for $\epsilon \leq \frac{1}{8}$.*

Proof (Sketch). By assumption, the probability that in a single run the outcome of processor i $P[i] \neq r$ (resp., $P[i] \neq x - r$), for all i, is at most ϵ. Thus, both calls to P return a correct outcome with probability at least $1-2\epsilon$. Letting $\alpha = 1-2\epsilon$, and adjusting the number of runs so that this happens for all processors, the claim follows from Proposition 1. □

3.4 Self-tester/corrector for agreement: parallel mode

We now consider the case where the network is not reset between calls, namely, where faults are persistent. The problem in this case is that if a protocol is prone to behave badly at processor i and processor i fails, then the bad behavior of the protocol is "shielded" by the processor failure. We need a mechanism to unshield the protocol behavior, so that the remaining correct processors have enough information to assess its quality.

To this end we suggest the technique we call *ID re-assignment.* The goal of the technique is to enable each and every remaining processor to "impersonate" every other processor in the network. Roughly, the implementation goes as follows. We do n parallel executions of the tester (resp. corrector) presented before, where each execution represents a different cyclic shift of the processors' IDs. This implies that processor 1 runs the protocol as itself, as processor 2, and so on, impersonating all other processors. Each processor collects a decision value for every impersonated ID. A unanimous decision by all impersonated

IDs determines whether the protocol will pass the test (or output the corrected value). Since each run may fail with negligible inverse exponnential probability, the combined protocol still succeeds with overwhelming probability.

Remark 1. ID re-assignment can be augmented to a *randomized* ID re-assignment via a shared random coin procedure. In environments with benign failures, such as the ones considered in this section, a shared random coin protocol is relatively easy to design (see, e.g., [5, 13]). Using the shared coin procedure, processors can draw random permutations of their IDs and execute an adequate number of paralle runs under these re-assignments. This allows to self-test/correct under more powerful network adversaries, such as "biased" (asymmetric) or arbitrary (in the latter case, the adversary does not have access to the shared coin).

4 Privacy-Preserving Self-Correctors

The above already teaches us how the design of a self-testing/correcting pair has to take into consideration certain constraints. For example, it can only assume basic utility library routines to be correct. In the domain of distributed protocols other constraints may be posed on the problem and the addition of co-routines should not violate these requirements. This is particularly important in the case of secure protocols, where the design of self-testers and self-correctors should not create new security vulnerabilities.

In this section we consider this aspect versus a *passive* adversary (i.e., the "private" or "gossip failure" setting). We assume the setting of secure distributed computing against an all-powerful adversary. A number of researchers, beginning with Ben-Or *et al.* and Chaum *et al.* [9, 12], have designed distributed protocols for this setting. Informally, a protocol is said to be t-private if an adversary viewing the transcripts of any t parties learns nothing about the computation beyond what can be inferred from their t inputs together with the output.

Following Definition 4, we say that a self-tester (resp., self-corrector) for P is *privacy-preserving* if $\mathcal{T}_f^P$ (resp., $\mathcal{C}_f^P$) is private (with overwhelming probability) whenever P is private (with overwhelming probability).

Assume that the n-ary function f is computed over a finite field F, and can be represented as a polynomial of degree d over F.[1] Let F^n denote the standard n-dimensional vector space over F. In this section we assume that random choices are made according to U_{F^n}, the uniform probability distribution on F^n. We assume, by the library paradigm of [11], that we have reliable programs to compute f *locally.* This is a reasonable assumption, as typically the task of a distributed protocol is conceptually simple, and the whole point is to be able to cope with the unpredictability of its environment.

A generic self-tester for P is given in Figure 4. The protocol simply verifies the local computation of f with the distributed execution of P at a number

[1] Most of the protocols performing secure/private computations are—or can be tuned to be—over a finite field. Also, any function over a finite field can be represented as a polynomial.

of randomly chosen inputs. The correctness of the self-tester is clear. Here we concentrate on the privacy issue, so we leave the constants unspecified—but assume that they satisfy the Chernoff bounds.

Our privacy-preserving self-corrector for P is given in Figure 5. It uses the "low-degree polynomial technique" of Beaver and Feigenbaum [7] to transform a function call at the desired (non-random) input into a collection of function calls at uniformly random inputs. The following series of lemmas establish that the self-corrrector is privacy-preserving.

Protocol Private_Self_Tester$(f, P, , \epsilon, \beta)$

$N \leftarrow O(\frac{\ln(n/\beta)}{\epsilon})$
$S \leftarrow 0$
for $m = 1, \cdots, N$ (in parallel)
{ choose $x^{(m)} \in_{U F^n} F^n$;
compute *locally* $f(x^{(m)})$
$P(x^{(m)})$
if $f \neq P$
$S \leftarrow S + 1$
}
if $S/N > \epsilon/Const$
"FAIL"
else "PASS"

Fig. 4. Generic self-tester for privacy; code for proc. i

Lemma 2. *If all oracle calls to P are correct, then the self-corrector of Figure 5 outputs $f(x)$.*

Proof (Sketch). By the same construction as in a theorem in [7], it follows that the value returned by $P(v^{(k)})$ corresponds to $g(k)$ for some univariate polynomial g of degree at most d. By the same construction, we know that $g(0) = f(x)$. □

Lemma 3. *If at most one call to the oracle is not t-private, then the self-corrector is t-private.*

The proof follows from the pairwise independence of the $v_i^{(k)}$. Note the importance of performing the interpolation of g from the shares of the $\{w_i\}_i$. Those intermediate values might leak information about the $v_i^{(k)}$ (and hence the x_i) even if the protocol P is perfectly private, depending on the nature of the underlying function f being computed by P.

Protocol Privacy_Preserving_Self_Corrector(P, x, ϵ, β)

```
N ← O(ln(n/β)/ε)
for m = 1, ···, N
{    choose α_i ∈_U F
     for k = 1, ···, d + 1
          v_i^(k) ← α_i k + x_i
     for k = 1, ···, d + 1 (in parallel)
          P(v^(k)) (yielding the share of w_k)
     interpret w_k as g(k) for some function g of degree d
     interpolate the function g at the points 1, 2, ..., d + 1
          using the shares of w_1, w_2, ..., w_{d+1}
     answer_m ← (share of) g(0)
}
P[i] ← most common answer in answer_m, 1 ≤ m ≤ N
```

Fig. 5. Privacy-preserving self-corrector; code for proc. i

Theorem 3. *The protocol of Figure 5 is a privacy-preserving* $\frac{\epsilon}{(d+1)}$*-self-corrector for any function* f *of degree* d *(with respect to the distribution* U_{F^n}*).*

Proof (Sketch). Because $\mathtt{error}(f, P, U_{F^n}, \mathcal{E}) \leq \frac{\epsilon}{(d+1)}$, all $(d+1)$ outputs of P are correct with probability at least $1-\epsilon$ each time through the loop. If all $(d+1)$ outputs are correct we know by Lemma 2 that $\text{answer}_m = f(x)$. Chernoff bounds provide the desired confidence. t-privacy follows from Lemma 3. □

5 Conclusions

This work has initiated the area of *self-testing/correcting protocols*, which gives a methodology to add reliability on-line and *in situ* to protocols by adding simple co-routines. In [11] and its extensions numerous numerical functions have been shown to have sequential self-testers and self-correctors. Applying the library paradigm one can assume that the correct sequential program is available, and use it to design a generic tester for the distributed version of the same numerical function (details deferred to the full version).

Acknowledgements. The authors thank Alain Mayer for his contributions during the initial stages of this research.

References

1. Y. Afek, S. Kutten and M. Yung, "The Local Detection Paradigm and Its Applications to Self-Stabilization." *TCS* 186 (1-2), pp. 199-229 (1997).

2. L. Adleman and K. Kompella. "Fast Checkers for Cryptography." In *CRYPTO '90*, pp. 515-529.
3. B. Awerbuch, B. Patt-Shamir and G. Varghese, "Self Stabilizing by local Checking and Correction." In *FOCS '91*, pp. 268-277.
4. M. Blum, "Designing programs to check their work." *Communications of the ACM.*
5. P. Berman and A. Bharali, "Quick Atomic Broadcast." In *WDAG '93.*
6. M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. "Checking the Correctness of Memories." In *FOCS '91*, pp. 90-99.
7. D. Beaver and J. Feigenbaum, "Hiding Instances in Multioracle Queries." In *STACS '90*, pp. 37-48.
8. M. Bellare, J. Garay and T. Rabin. "Batch Verification with Applications to Cryptography and Checking" (invited paper). In *LATIN '98*, pp. 170-191.
9. M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation." In *STOC '88*, pp. 1-10.
10. M. Blum and S. Kannan, "Designing programs to check their work." In *STOC '89*, pp. 86-97.
11. M. Blum, M. Luby, and R. Rubinfeld, "Self-Testing/Correcting with Applications to Numerical Problems." In *STOC '90*, pp. 73-83.
12. D. Chaum, C. Crepeau and I. Damgard. "Multiparty Unconditionally Secure Protocols." In *STOC '88*, pp. 11-19.
13. B. Chor, M. Merritt, and D. Shmoys, "Simple Constant-Time Consensus Protocols in Realistic Failure Models." In *PODC '85*, pp. 152-162.
14. D. Dolev, "The Byzantine Generals Strike Again," *Journal of Algorithms*, 3(1):14-30, January 1982.
15. S. Dolev, A. Israeli and S. Moran, "Self Stabilization of Dynamic Systems assuming only Read/Write Atomicity." In *PODC '90*, pp. 103-117.
16. C. Dwork and Y. Moses, "Knowledge and common knowledge in a Byzantine environment: crash failures." Information and Computation **88**, pp. 156-186 (1990).
17. F. Ergun, S. Ravi Kumar and R. Rubinfeld. "Approximate Checking of Polynomials and Functional Equations." In *FOCS '96*, pp. 592-601.
18. Y. Frankel, P. Gemmell and M. Yung. "Witness-Based Cryptographic Program Checking and Robust Function Sharing." In *STOC '96*, pp. 499-508.
19. P. Gemmell, R. Lipton, R. Rubinfeld, M. Sudan, and A. Wigderson. "Self-testing/correcting for polynomials and for approximate functions." In *STOC '91*, pp. 32-42.
20. S. Kannan, "Program Checkers for Algebraic Problems." Ph.D. Thesis, ICSI TR-89-064, December 1989.
21. R. Karp, M. Luby, and N. Madras, "Monte-Carlo Approximation Algorithms for Enumeration Problems." *Journal of Algorithms* **10**, pp. 429-448 (1989).
22. S. Katz and K. Perry, "Self-Stabilizing Extensions for Message-Passing Systems." In *PODC '90*, pp. 91-101.
23. R. Lipton, "New Directions in Testing," DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 2 (1991), pp. 191-202.
24. L. Lamport, R.E. Shostak and M. Pease, "The Byzantine Generals Problem," *ACM ToPLaS*, Vol. 4, No. 3 (1982), pp. 382-401.
25. R. Rubinfeld, "A Mathematical Theory of Self-Checking, Self-Testing and Self-Correcting Programs." Ph.D. Thesis, ICSI TR-90-054, October 1990.
26. R. Rubinfeld. "On the Robustness of Functional Equations." In *FOCS '94*, pp. 2-13.
27. R. Rubinfeld. "Designing Checkers for Programs that Run in Parallel." Algorithmica 15(4), pp. 287-301, 1996.