Self-Testing/Correcting with Applications to Numerical Problems

Manuel Blum [*] Michael Luby [†] Ronitt Rubinfeld [‡]

Abstract

Suppose someone gives us an extremely fast program P that we can call as a black box to compute a function f. Should we trust that P works correctly? A *self-testing/correcting pair* for f allows us to: (1) estimate the probability that $P(x) \neq f(x)$ when x is randomly chosen; (2) on *any* input x, compute $f(x)$ correctly as long as P is not too faulty on average. Furthermore, both (1) and (2) take time only slightly more than the original running time of P.

We present general techniques for constructing simple to program self-testing/correcting pairs for a variety of numerical functions, including integer multiplication, modular multiplication, matrix multiplication, inverting matrices, computing the determinant of a matrix, computing the rank of a matrix, integer division, modular exponentiation and polynomial multiplication.

1 Introduction

Consider the task of writing a program P to evaluate a function f. One of the main difficulties is that when P is implemented it is difficult to verify that $P(x) = f(x)$ for all inputs x. There are two traditional approaches to this problem, program verification and program testing. Program verification has had fairly limited success because even relatively simple programs are hard to prove correct. Furthermore, even if the proof is correct it only makes a statement about the program as it is written on paper, not about the compiled code nor about the hardware on which it runs. Traditional testing has two drawbacks. First, the test inputs typically do not cover all inputs encountered when the program is actually used, and thus on a particular input the user has no guarantee that the program output is correct. Second, often during testing another program P' is used to compute f to compare against the answer of P, and thus there is a reliance on the correctness of another program P' that is in no quantifiable way different than the program P it is being used to test.

[*]Computer Science Division, U.C. Berkeley, Berkeley, California 94720, research partially supported by NSF Grant No. CCR 88-13632.

[†]International Computer Science Institute, Berkeley, California 94704, research partially supported by NSF operating grant CCR-9016468 and by grant No. 89-00312 from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel.

[‡]Computer Science Department, Hebrew University, Israel, research done while at UC Berkeley and partially supported by an IBM Graduate Fellowship and NSF Grant No. CCR 88-13632.

The theory of *result checking*, introduced in [10, Blum], provides an attractive supplement to traditional approaches for verifying that a program is correct (see also [12, Blum Kannan], [11, Blum Raghavan], [24, Kannan]). Informally, the idea is to write a very simple program C, called the result checker, which is to be run in conjunction with P to verify that $P(x) = f(x)$ in the following sense. If P is correct for all inputs (and thus also $P(x) = f(x)$) then the result checker outputs "PASS", but if $P(x) \neq f(x)$ then the result checker outputs "FAIL".[1] The result checker C may call P on inputs other than x, but it may only access P as a black box, and does not have access to the program code of P. The result checker C is written for a specific function f, but C must work for all programs P that purport to compute f.

1.1 Self-Testing/Correcting

Although a result checker can be used to verify whether $P(x) = f(x)$, it does not give a method for computing the correct answer in the case that P is found to be faulty. In this paper[2], we introduce the theory of *self-testing/correcting*, which is an extension of the theory of result checkers that is meant to address this issue. Intuitively, a probabilistic program T_f is a *self-tester* for f if, for any program P that supposedly computes f, T_f can make calls to P to estimate the probability that $P(x) \neq f(x)$ for a random input x. We call this probability the error probability of P. A probabilistic program C_f is a *self-corrector* for f if, for any program P such that the error probability of P is sufficiently low, for any input x, C_f can make calls to P to compute $f(x)$ correctly. Thus, the advantage of self-testing/correcting over result checkers is that on a given input x a result checker only verifies that $P(x) = f(x)$, whereas a self-corrector can be used to compute $f(x)$ correctly making calls to P, even in the case when $P(x) \neq f(x)$, as long as P is verified to be correct for most inputs using the self-tester.

The question remains, how to verify that the self-testing/correcting pair meets its specifications. Although there is no final answer to this question, there are some partial answers. First, it has been our experience that the code for the self-testing/correcting pairs we have designed is often much simpler than reasonably fast programs for computing f directly, and is therefore more likely to be correct on these grounds alone. Moreover, a lot of time can be spent in the design of a self-testing/correcting pair to try and ensure that it is correct, because a self-testing/correcting pair can be used on all revisions in the future to the currently used program P for computing f. In the case of result checkers, [Blum 88] suggests that the result checker should be in some quantifiable way "different" than any program P that correctly computes f directly, because then it is unlikely that the result checker makes mistakes of the same type as those made by P. We adopt this same philosophy, and require that our self-testing/correcting pair be "different" in the following sense. We call the running time of T_f, not counting the time for calls to the program P, the *incremental time* of T_f. We say that T_f is *different* if the incremental time of T_f is faster than the running time for any correct program for computing f directly. Analogous definitions apply to C_f. We insist that both T_f and C_f be different, which ensures that the self-testing/correcting pair is doing something quantifiably different than computing f directly, because there is not enough time for this.

[1]Note that there is no specification of the behavior of C when $P(x) = f(x)$ but P incorrectly computes f on some inputs other than x. The "natural" requirement would be that in this case the output of C is "PASS". However, this can be easily shown to imply that C has to correctly compute $f(x)$ on its own without any calls to P, which is clearly not in the spirit of allowing C to be a much simpler program than any correct program for f.

[2]A preliminary version of this paper appeared in [14, Blum, Luby, Rubinfeld].

We call the running time of T_f, counting the time for calls to the program P, the *total time* of T_f. We say that T_f is *efficient* if the total time is linear in the running time of P. We insist that both T_f and C_f be efficient, which ensures that the advantages we gain by using the self-testing/correcting pair are not overwhelmed by an inordinate running time slowdown.

A self-testing/correcting pair (T_f, C_f) for a function f is a powerful tool. A user can take *any* program P that supposedly computes f and self-test it with T_f. If P passes the self-test then, on any input x, the user can call C_f, which in turn makes calls to P, to correctly compute $f(x)$. Even a program P that computes f incorrectly for a small but significant fraction of the inputs can be used with confidence to correctly compute $f(x)$ for any input x. In addition, if in the future somebody designs a faster program P' for computing f, then the same pair (T_f, C_f) can be used to self-test/correct P' without any further modifications. Thus, it makes sense to spend a reasonable amount of time designing self-testing/correcting pairs for functions commonly used in practice and for which a lot of effort is spent writing super-fast programs. For example, integer multiplication and matrix multiplication are commonly used functions for which fast but complicated programs have been written and implemented ([17, Coppersmith Winograd], [38, Strassen], [35, Schönhage], [36, Schönhage Strassen]). Thus, the self-testing/correcting pairs we develop may be useful in practice.

We develop general techniques for constructing simple to program self-testing/correcting pairs for a variety of numerical functions. Roughly speaking, we develop techniques to correct random self-reducible functions, to test linear functions and to test by bootstrapping functions that are both random self-reducible and downward self-reducible. Our techniques apply to integer multiplication, the mod function, modular multiplication, integer division, polynomial multiplication, modular exponentiation, matrix multiplication, determinant, matrix inversion and matrix rank. For all of these functions, except for modular exponentiation in the case when the factorization of the modulus is not known, the incremental time is linear in the input size and the total time is linear in the running time of P. Thus, for these functions, the self-testing/correcting pair is both different and efficient. For modular exponentiation in the case when the factorization of the modulus is not known, the self-testing/correcting pair is different and close to efficient.

The theory of self-testing leads to interesting mathematical questions about properties that characterize a function. We show that certain properties that characterize a function which hold on *every* input can be replaced by the same property which only holds for a *large fraction* of inputs. For example, suppose f is a function that maps a group G to a group H. We say that f is *linear* if, for *all* x and y in G, $f(x +_G y) = f(x) +_H f(y)$ (where $+_G$ and $+_H$ are the group operations over G and H, respectively). The results in Section 4 relax the condition required for linearity in the following sense: they show that if, for a *large fraction* of x, y, $f(x +_G y) = f(x) +_H f(y)$, then there is a linear function g such that $f(x)$ is equal to $g(x)$ for most x. Thus f is still essentially a linear function. [21, Gemmell Lipton Rubinfeld Sudan Wigderson] shows that a similar property and relaxation holds for polynomials. Since it is computationally much easier to determine whether a property is satisfied most of the time than it is to determine whether it is always satisfied, this relaxation is important for self-testing.

1.2 Libraries

Often programs for related functions are grouped in packages; common examples include packages that solve statistics problems or packages that do matrix manipulations. We extend the theory proposed in [10, Blum] to allow the use of several programs, or a *library*, to aid in self-testing and self-correcting. We show that the library approach allows one to construct self-testing/correcting pairs for functions which did not previously have efficient self-testing or self-correcting programs, or even result checkers. Working with a library of programs rather than with just a single program is a key idea: enormous difficulties arise in attempts to design a self-testing/correcting pair for the determinant or for computing the rank in the absence of programs for matrix multiplication and inverse.

The typical situation where the library approach is useful is for a function f where the natural way to compute f is by making a small number of calls to another function f', where the running time to compute f' is of the same order of magnitude as the running time to compute f. The running time to compute f' makes it hard to design a self-testing/correcting pair for f based on making calls to f' that is different. However, if it is possible to design a self-testing/correcting pair for f' that is different, then we can use the approach described in the next paragraph to design a self-testing/correcting pair for f.

The idea of the library approach is to first design a self-testing/correcting pair (T', C') for f', and then to design a self-testing/correcting pair (T, C) for f that makes calls to C' to compute f' instead of computing f' directly. Let P be the program that supposedly computes f and let P' be the program that supposedly computes f'. The incremental time of T is the running time of T, not counting calls to either C' or P, plus the incremental time for C' multiplied by the number of times C' is called. The total time of T counts the time for all calls to P and C', and within C' counts the time for calls to P'. The incremental and total times of C are defined analogously. The way the library of self-testing/correcting pairs is used in this example is as follows: First, T' tests that P' is not too faulty. Then, T tests that P is not too faulty. T makes calls to C', which in turn makes calls to P'. Finally, C computes f by making calls to both P and C', which in turn makes calls to P'. The properties are that if either P or P' is too faulty, then one of T' or T will output "FAIL", whereas if both P and P' are not too faulty, then C correctly computes f on all inputs with high probability.

1.3 Related Work and Extensions

[15, Blum Micali] construct a pseudo-random generator, where a crucial ingredient of the construction can be thought of as a self-correcting program for the discrete log function. [31, Rubinfeld] introduces result checking for parallel programs, and uses self-testing to design a constant depth circuit to check the majority function. A self-testing/correcting pair for a function f implies a result checker for f. A result checker for f implies a self-tester for f, but it is not known whether a result checker also implies a self-corrector. Previous to our work, [23, Kaminski] gives result checkers for integer and polynomial multiplication. Independently of our work, [1, Adleman Huang Kompella] give result checkers for integer multiplication and modular exponentiation. Both of these papers use very different techniques than ours. Previous to our work, [20, Freivalds] introduces a result checker for matrix multiplication over a finite field. We make use of this result checker when designing the self-testing/correcting pair for matrix multiplication over a finite field.

[26, Lipton], independently of our work, discusses the concept of self-correcting programs and for several functions uses it to construct a testing program with respect to any distribution *assuming that the programs are not too faulty with respect to a particular distribution.* To highlight the importance of being able to self-test, consider the mod function. To self-correct on input x and modulus R, the assumption in [26, Lipton] and here is that the program is correct for most inputs x *with respect to the particular modulus* R. This requires a different assumption for each distinct modulus R. Our self-testing algorithm for the mod function on input R can be used to efficiently either validate or refute this assumption.

Previously, [24, Kannan] provides an elegant result checker for computing the determinant of a matrix, but it is not efficient. Our self-correcting/testing pair for determinant is efficient, but it relies heavily on allowing the pair to call a library of linear algebra programs instead of restricting calls to a single program that supposedly computes determinant.

In this paper, we assume that the program's answer on a particular input does not depend on previous inputs. [13, Blum Luby Rubinfeld] considers the case when the program adaptively decides its answer based on previous inputs.

Recently, [16, Cleve Luby] have shown how to use these techniques to design a self-testing/-correcting pair for the trigonometric functions sin and cosin. [21, Gemmell Lipton Rubinfeld Sudan Wigderson] extend these results to design a self-testing/correcting pair for any polynomial function over finite fields and [34, Rubinfeld Sudan] extend this to polynomial functions over rational domain. [33, Rubinfeld] introduces an extension of this theory to the case when the program's answer is considered correct when it is a good approximation to the actual value of the function, and designs a self-testing/correcting pair for the quotient function.

The techniques in this paper have been applied to the theory of interactive proofs (see [22, Goldwasser Micali Rackoff], [3, Babai] and [9, Ben-Or Goldwasser Kilian Wigderson] for the discussion of interactive proofs). [27, Nisan] uses the self-testing/correcting technique based on bootstrapping developed in Section 7 and the observation about the permanent problem in [26, Lipton] (which is based on [6, Beaver Feigenbaum]) to construct a two-prover interactive proof system for the permanent problem, which led to the eventual discovery that *IP* = *PSPACE* ([19, Fortnow Karloff Lund Nisan], [37, Shamir], [4, Babai]).

The results in this paper are related to those in [5, Babai Fortnow Lund]. In order to show that the multi-prover version of *IP* is equal to *NEXPTIME*, [5, Babai Fortnow Lund] give a test for verifying that a given program P, which depends on n input variables, computes a function which is usually equal to some multi-linear function f of the n variables. Their results can be viewed as providing a self-tester for multi-linear multi-variate functions, assuming the ability to correctly compute linear functions of one variable. Combining their results with the self-testers for linear functions of one variable given in this paper yields a much simpler self-tester for multi-linear multi-variate functions, which uses only additions, comparisons and calls to P, and which is "different" in the sense used in this paper.

2 The Basics

DEFINITION 2.1 (distribution on inputs) *For expository purposes, we restrict ourselves to the case when f is a function of one input from some universe $\mathcal{I}$. Let $\mathcal{I}_1, \mathcal{I}_2, \ldots$ be a sequence of subsets of $\mathcal{I}$ such that $\mathcal{I} = \cup_{n \in \mathcal{N}} \mathcal{I}_n$. The subscript n indicates the "size" of the input to the function. Let $\mathcal{D} = \{\mathcal{D}_n | n \in \mathcal{N}\}$ be an ensemble of probability distributions such that $\mathcal{D}_n$ is a distribution on $\mathcal{I}_n$.*

DEFINITION 2.2 (error) *Let P be a program that supposedly computes f. Let* $\text{error}(f, P, \mathcal{D}_n)$ *be the probability that $P(x) \neq f(x)$ when x is randomly chosen in $\mathcal{I}_n$ according to $\mathcal{D}_n$.*

DEFINITION 2.3 (probabilistic oracle program) *A probabilistic program M is an* oracle *program if it makes calls to another program that is specified at run time. We let M^A denote M making calls to program A.*

DEFINITION 2.4 (self-testing program) *Let $0 \leq \epsilon_1 < \epsilon_2 \leq 1$. An* (ϵ_1, ϵ_2)-self-testing program *for f with respect to $\mathcal{D}$ is a probabilistic oracle program T_f that has the following properties for any program P on input n and confidence parameter $\beta > 0$.*

1. *If* $\text{error}(f, P, \mathcal{D}_n) \leq \epsilon_1$ *then T_f^P outputs "PASS" with probability at least $1 - \beta$.*
2. *If* $\text{error}(f, P, \mathcal{D}_n) \geq \epsilon_2$ *then T_f^P outputs "FAIL" with probability at least $1 - \beta$.*

To simplify the code somewhat, we make the convention that the self-tester immediately halts once it outputs an answer, either "PASS" or "FAIL". The value of ϵ_1 should be as close as possible to ϵ_2 to allow as faulty as possible programs P to pass test T_f^P and still have self-corrector C_f^P work correctly.

DEFINITION 2.5 (self-correcting program) *Let $0 \leq \epsilon < 1$. An* ϵ-self-correcting program *for f with respect to $\mathcal{D}$ is a probabilistic oracle program C_f that has the following property on input n, $x \in \mathcal{I}_n$ and $\beta > 0$. If* $\text{error}(f, P, \mathcal{D}_n) \leq \epsilon$ *then $C_f^P(x) = f(x)$ with probability at least $1 - \beta$.*

DEFINITION 2.6 (self-testing/correcting pair) *A* self-testing/correcting pair *for f is a pair of probabilistic programs (T_f, C_f) such that there are constants $0 \leq \epsilon_1 < \epsilon_2 \leq \epsilon < 1$ and an ensemble of distributions $\mathcal{D}$ such that T_f is an (ϵ_1, ϵ_2)-self-testing program for f with respect to $\mathcal{D}$ and C_f is an ϵ-self-correcting program for f with respect to $\mathcal{D}$.*

DEFINITION 2.7 (running time) *Let M^P be a probabilistic oracle machine M making oracle calls to P. The* incremental time *of M^P is the maximum over all inputs x of length n of the running time of $M^P(x)$, not counting the time for calls to P. The* total time *of M^P is the maximum over all inputs x of length n of the running time of $M^P(x)$, counting the time for calls to P.*

DEFINITION 2.8 (different) *We say that self-testing/correcting pair (T_f, C_f) is* different *if, for all programs P, the incremental time of both T_f^P and C_f^P is smaller than the running time of the fastest known program for computing f directly.*

DEFINITION 2.9 (efficient) *We say that self-testing/correcting pair (T_f, C_f) is* efficient *if, for all programs P, the total time of both T_f^P and C_f^P is linear in the running time of P and the input size.*

We insist that a self-testing/correcting pair be both different and efficient, although for modular exponentiation when the factorization of the modulus is unknown we are forced to relax the efficiency requirement somewhat. In the definitions of *different* and *efficient*, we ignore the running time dependence on the confidence parameter β, which is typically a multiplicative factor of $O(\ln(1/\beta))$.[3]

[3]In this paper, $\ln \alpha$ denote the natural log of α. In some cases, $\ln \alpha$ is to be thought of as an integer, in which case it is the least integer greater than or equal to $\ln \alpha$.

Because self-testers must be *different*, the strategy used by T_f^P cannot be the naive technique of choosing $x \in \mathcal{I}_n$ according to $\mathcal{D}_n$ and seeing if $P(x) = f(x)$, because this requires computation of $f(x)$. Similarly, C_f^P cannot simply call P on input x and hope that $P(x) = f(x)$, because P is allowed to be faulty on a fraction of the inputs, and in particular it might be faulty on input x. In many of the self-testers and self-correctors we design, we exploit the ability to compute $f(x)$ indirectly by computing f on random inputs. This property is explained in the following definition.

Definition 2.10 (random self-reducibility property) *Let $x \in \mathcal{I}_n$. Let $c > 1$ be an integer. We say that f is* c-random self-reducible *if $f(x)$ can be expressed as an easily computable function F_{random} of x, $a_1, \ldots, a_c$ and $f(a_1), \ldots, f(a_c)$, where $a_1, \ldots, a_c$ are easily computable given x and each a_i is randomly distributed in $\mathcal{I}_n$ according to $\mathcal{D}_n$.*[4] *By* easily, *we mean that the total computation time of the random self-reduction is smaller than that of computing $f(x)$.*

One of the strengths of this property is that it can be used to transform a program that is correct on a large enough fraction of the inputs into a program that computes $f(x)$ correctly with high probability for *any* input x.

Many of the functions we consider are on the integers or on initial segments of the integers. We often use the following notation.

Definition 2.11 (arithmetic notation) *For any positive integer R, let $\mathcal{Z}_R$ denote the set of integers $\{0, \ldots, R-1\}$, let $+_R$ denote integer addition mod R and let $\cdot_R$ denote integer multiplication mod R. Let $\mathcal{Z}_R^* = \{x \in \mathcal{Z}_R : \gcd(x, R) = 1\}$.*

For simplicity, in the description of all of our self-correcting/testing programs we omit the following simple but crucial piece of the code.

Definition 2.12 (range-check code) *Whenever the self-corrector or self-tester makes a call to P, it verifies that the answer returned by P is in the proper range, e.g. for $f(x, R) = x \bmod R$ the proper range is $\mathcal{Z}_R$. If the answer is not in the proper range, then the program resets the answer to a default value in the range, e.g. for $f(x, R) = x \bmod R$, the default value could be 0.*

The range-check code in effect modifies the original P into a modified P. However, the modified P is at least as correct for computing f as the original P. For correctness, it is crucial that the self-tester and the self-corrector both use the same default value in the range-check code. This is because we want the self-corrector and self-tester to be calling as an oracle the same P. In most cases, the range-check code is straightforward, and we discuss it in those cases where it is not.

We often consider uniform probability distributions on sets. Thus, we introduce the following notation.

Definition 2.13 (uniform probability distribution) *For any set X, let $\mathcal{U}_X$ denote the uniform probability distribution on X. For example, $\mathcal{U}_{\mathcal{Z}_{2^n}}$ is the uniform distribution on $\mathcal{Z}_{2^n}$, whereas $\mathcal{U}_{\{R\}}$, where R is a positive integer, is the probability distribution such that R has probability one. We let $x \in_{\mathcal{U}} X$ denote that x is randomly and uniformly distributed in X.*

[4] However, no independence between these random variables is needed, e.g. given the value of a_1 it is not necessary that a_2 be randomly distributed in $\mathcal{I}_n$ according to $\mathcal{D}_n$.

3 Self-Correcting

In this section, we describe self-correctors for a variety of numerical functions. We start with self-correcting because the self-correctors for our applications are much more intuitive than the corresponding self-testers, and in addition the self-correctors are substantially easier to prove correct.

In the following subsections, we show the specific details of the self-correcting programs for the mod function, integer multiplication, modular multiplication, modular exponentiation, integer division, matrix multiplication and polynomial multiplication. All of these self-correcting programs follow the same outline and rely on the random self-reducibility property (defined in the preceding subsection) of the given function. In Subsection 3.8 we give a self-correcting program that works for any random self-reducible function. [26, Lipton] uses the same basic outline to develop a self-correcting program for any polynomial over a finite field.

3.1 Mod Function

We consider computing an integer mod$R$ for a positive number R. In this case, $f(x,R) = x \bmod R$. Assume that we have a program P such that $\text{error}(f, P, \mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\{R\}}) \leq 1/8$. The following program is a 1/8-self-correcting program for f making oracle calls to P with respect to $\mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\{R\}}$. The input to the program is n, R, $x \in \mathcal{Z}_{R2^n}$ and the confidence parameter β.

Program Mod Function Self-Correct(n, R, x, β)

```
    $N \leftarrow 12\ln(1/\beta)$
    Do for $m = 1, \ldots, N$
      Call **Random_Split**$(R2^n, x, x_1, x_2, c)$
      $answer_m \leftarrow P(x_1, R) +_R P(x_2, R)$
    Output the most common answer among $\{answer_m : m = 1, \ldots, N\}$
```

Function Random_Split (M, z, z_1, z_2, e)

```
    Choose $z_1 \in_{\mathcal{U}} \mathcal{Z}_M$
    If $z_1 \leq z$ then $e \leftarrow 0$ else $e \leftarrow 1$
    $z_2 \leftarrow eM + z - z_1$
```

We need the following proposition in the proof of correctness of this and many subsequent programs.

Proposition 1 *Let $x_1, \ldots, x_m$ be independent 0/1 valued random variables such that for each $i = 1, \ldots, m$, $\Pr[x_i = 1] \geq 3/4$. Then,*

$$\Pr\left[\sum_{i=1}^{m} x_i > m/2\right] \geq 1 - e^{-m/12}.$$

PROOF: Use standard Chernoff bounds. ■

Lemma 2 *The above program is a 1/8-self-correcting program for the mod function.*

PROOF: For $i \in \{1,2\}$, $x_i \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$. Thus, by the properties of P, $P(x_i, R) \neq x_i \bmod R$ with probability at most $1/8$, and consequently both calls to P in a single loop return the correct answer with probability at least $3/4$. Because $x = x_1 + x_2 - cR2^n$, $x \bmod R = x_1 \bmod R +_R x_2 \bmod R$. Thus, if both calls to P are correct, $answer_m = x \bmod R$. The lemma follows from Proposition 1. ■

The mod function self-correcting program is very simple to code, the only operations used are integer additions, comparisons and calls to the program P. This is true because in the computation of $answer_m$ because of the implicit range-check code (see page 7), both $P(x_1, R)$ and $P(x_2, R)$ are in $\mathcal{Z}_R$. Thus, to compute $P(x_1, R) +_R P(x_2, R)$ consists of one integer addition, one comparison and possibly one subtraction. Note that the self-correcting program is different because the incremental time is linear in n, and it is also efficient, because the total time is linear in the running time of P.

3.2 Integer Multiplication

For integer multiplication, $f(x,y) = x \cdot y$. Suppose that both x and y are in the range $\mathcal{Z}_{2^n}$ for some positive integer n. Assume that we have a program P such that $\text{error}(f, P, \mathcal{U}_{\mathcal{Z}_{2^n}} \times \mathcal{U}_{\mathcal{Z}_{2^n}}) \leq 1/16$. The following program is a 1/16-self-correcting program for f making oracle calls to P with respect to $\mathcal{U}_{\mathcal{Z}_{2^n}} \times \mathcal{U}_{\mathcal{Z}_{2^n}}$. The input to the program is n (the length of the inputs), $x, y \in \mathcal{Z}_{2^n}$ (the numbers to be multiplied together) and the confidence parameter β.

Program Integer Multiplication Self-Correct(n, x, y, β)

```
$N \leftarrow 12\ln(1/\beta)$
Do for $m = 1, \ldots, N$
  Call **Random_Split**$(2^n, x, x_1, x_2, c)$
  Call **Random_Split**$(2^n, y, y_1, y_2, d)$
  $answer_m \leftarrow P(x_1,y_1) + P(x_1,y_2) + P(x_2,y_1) + P(x_2,y_2) - cy2^n - dx2^n - cd2^{2n}$
Output the most common answer among $\{answer_m : m = 1, \ldots, N\}$
```

Lemma 3 *The above program is a 1/16-self-correcting program for integer multiplication.*

PROOF: For $i, j \in \{1,2\}$, the pair $(x_i, y_j) \in_{\mathcal{U}} \mathcal{Z}_{2^n} \times \mathcal{Z}_{2^n}$. Thus, by the properties of P, $P(x_i, y_j) \neq x_i \cdot y_j$ with probability at most $1/16$, and consequently all four calls to P in a single loop return the correct answer with probability at least $3/4$. Because $x = x_1 + x_2 - c2^n$ and $y = y_1 + y_2 - d2^n$, $x \cdot y = x_1 \cdot y_1 + x_1 \cdot y_2 + x_2 \cdot y_1 + x_2 \cdot y_2 - cy2^n - dx2^n - cd2^{2n}$. Thus, if all four calls to P are correct, $answer_m = x \cdot y$. The lemma follows from Proposition 1. ■

The integer multiplication self-correcting program is very simple to code, the only operations used are integer additions, shifts, comparisons and calls to the program P.

3.3 Modular Multiplication

We now consider multiplication of integers $\bmod R$ for a positive number R. In this case, $f(x, y, R) = x \cdot_R y$. Suppose that both x and y are in the range $\mathcal{Z}_{R2^n}$ for some positive integer n. Assume that we have a program P such that error$(f, P, \mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\{R\}}) \leq 1/16$. The following program is a 1/16-self-correcting program for f making oracle calls to P with respect to $\mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\{R\}}$. The input to the program is R, $x, y \in \mathcal{Z}_{R2^n}$ and the confidence parameter β.

Program Modular Multiplication Self-Correct(R, x, y, β)

```
$N \leftarrow 12\ln(1/\beta)$
Do for $m = 1, \ldots, N$
  Call **Random_Split**$(R2^n, x, x_1, x_2, c)$
  Call **Random_Split**$(R2^n, y, y_1, y_2, d)$
  $answer_m \leftarrow P(x_1, y_1, R) +_R P(x_2, y_1, R) +_R P(x_1, y_2, R) +_R P(x_2, y_2, R)$
Output the most common answer among $\{answer_m : m = 1, \ldots, N\}$
```

Lemma 4 *The above program is a* 1/16*-self-correcting program for modular multiplication.*

PROOF: For $i, j \in \{1, 2\}$, the pair $(x_i, y_j) \in_{\mathcal{U}} \mathcal{Z}_{R2^n} \times \mathcal{Z}_{R2^n}$. Thus, by the properties of P, $P(x_i, y_j) \neq x_i \cdot y_j$ with probability at most 1/16, and consequently all four calls to P in a single loop return the correct answer with probability at least 3/4. Because $x = x_1 + x_2 - cR2^n$ and $y = y_1 + y_2 - dR2^n$, $x \cdot_R y = (x_1 \cdot_R y_1) +_R (x_1 \cdot_R y_2) +_R (x_2 \cdot_R y_1) +_R (x_2 \cdot_R y_2)$. Thus, if all four calls to P are correct, $answer_m = x \cdot_R y$. The lemma follows from Proposition 1. ■

3.4 Modular Exponentiation

We now consider exponentiation of integers $\bmod R$ for a positive number R. In this case, $f(a, x, R)$ $= a^x \bmod R$. We restrict attention to the case when $\gcd(a, R) = 1$ and when we know the factorization of R, and thus we can easily compute $\phi(R)$, where ϕ is Euler's function. Suppose that x is in the range $\mathcal{Z}_{\phi(R)2^n}$. Assume that we have a program P such that error$(f, P, \mathcal{U}_{\{a\}} \times \mathcal{U}_{\mathcal{Z}_{\phi(R)2^n}} \times \mathcal{U}_{\{R\}}) \leq 1/8$. The following program is a 1/8-self-correcting program for f making oracle calls to P with respect to $\mathcal{U}_{\{a\}} \times \mathcal{U}_{\mathcal{Z}_{\phi(R)2^n}} \times \mathcal{U}_{\{R\}}$. The input to the program is R, a, $x \in \mathcal{Z}_{\phi(R)2^n}$ and the confidence parameter β.

Program Modular Exponentiation Self-Correct (R, a, x, β)

```
$N \leftarrow 12\ln(1/\beta)$
Do for $m = 1, \ldots, N$
  Call **Random_Split**$(\phi(R)2^n, x, x_1, x_2, c)$
  $answer_m \leftarrow P(a, x_1, R) \cdot_R P(a, x_2, R))$
Output the most common answer among $\{answer_m : m = 1, \ldots, N\}$
```

Lemma 5 *The above program is a* 1/8*-self-correcting program for modular exponentiation.*

PROOF: For $i \in \{1, 2\}$, $x_i \in_{\mathcal{U}} \mathcal{Z}_{\phi(R)2^n}$. Thus, by the properties of P, $P(a, x_i, R) \neq a^{x_i} \bmod R$ with probability at most 1/8, and consequently both calls to P in a single loop return the correct answer with probability at least 3/4. Because $x = x_1 + x_2 - c\phi(R)2^n$, and because $\gcd(a, R) = 1$ implies that $a^{\phi(R)} = 1 \bmod R$, $a^x \bmod R = a^{x_1} \bmod R +_R a^{x_2} \bmod R$. Thus, if both calls to P are correct, $answer_m = a^x \bmod R$. The lemma follows from Proposition 1. ■

The modular exponentiation self-correcting program is very simple to code. The hardest operation to perform is the modular multiplication $P(a, x_1, R) \cdot_R P(a, x_2, R)$. The self-correcting program can compute this multiplication directly, but another alternative is to use the library approach described informally here and in more detail in a subsequent section.

Let f be the modular exponentiation function and let f' be the modular multiplication function. Let P be a program that supposedly computes f and let P' be a program that supposedly computes f'. Let C' be the modular multiplication self-correcting program described in a previous subsection and let C be the modular exponentiation self-correcting program just described. If $\text{error}(f, P, \mathcal{U}_{\{a\}} \times \mathcal{U}_{\mathcal{Z}_{\phi(R)2^n}} \times \mathcal{U}_{\{R\}}) \leq 1/8$ and if $\text{error}(f', P', \mathcal{U}_{\mathcal{Z}_R} \times \mathcal{U}_{\mathcal{Z}_R} \times \mathcal{U}_{\{R\}}) \leq 1/16$ then we can use C, making calls to P and making calls to C', which in turn makes to P', to self-correct f. Using this approach, the only operations computed by either C or C' are integer additions, comparisons and calls to the programs P and P'. The self-correcting program is different because the incremental time (which excludes the time for calls to both P and P') is linear in n, and it is also efficient, because the total time (which counts the time for calls to both P and P') is within a constant multiplicative factor of the running time of P assuming that P' runs at least as quickly as P.

3.5 Integer Division

We now consider division of integers by R for a positive number R. In this case, $f(x, R) = (x \text{ div } R, x \bmod R)$. Suppose that x is in the range $\mathcal{Z}_{R2^n}$. Assume that we have a program P such that $\text{error}(f, P, \mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\{R\}}) \leq 1/8$. The following program is a $1/8$-self-correcting program for f making oracle calls to P with respect to $\mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\{R\}}$. The input to the program is R, $x \in \mathcal{Z}_{R2^n}$ and the confidence parameter β.

We refer to the output of P as $P(x, R) = (P_{div}(x, R), P_{mod}(x, R))$.

Program Integer Division Self-Correct(R, x, β)

```
$N \leftarrow 12 \ln(1/\beta)$
Do for $m = 1, \ldots, N$
  Call **Random_Split**$(R2^n, x, x_1, x_2, c)$
  $Divans_m \leftarrow (P_{div}(x_1, R) + P_{div}(x_2, R)) + (P_{mod}(x_1, R) + P_{mod}(x_2, R)) \text{ div } R - c \cdot 2^n$
  $Modans_m \leftarrow P_{mod}(x_1, R) +_R P_{mod}(x_2, R)$
Output the most common answer among $\{(Divans_m, Modans_m) : m = 1, \ldots, N\}$
```

Lemma 6 *The above program is a $1/8$-self-correcting program for integer division.*

PROOF: Follows the outline of the proof of Lemma 2. ■

As in the self-corrector for the mod function, both the mod and div computed by the self-corrector are easy to code. This is true because in the computation of $Modans_m$, the range-check code (see page 7) ensures that both $P_{mod}(x_1, R)$ and $P_{mod}(x_2, R)$ are in $\mathcal{Z}_R$. Thus, to compute $P_{mod}(x_1, R) +_R P_{mod}(x_2, R)$ consists of one integer addition, one comparison and possibly one subtraction. In the computation of $Divans_m$, computing $(P_{mod}(x_1, R) + P_{mod}(x_2, R)) \text{ div } R$ consists of one integer addition and one comparison.

3.6 Matrix Multiplication

We consider multiplication of matrices over a finite field. Let $M_{n\times n}[F]$ be the set of $n \times n$ matrices over the finite field F. Then, for all $A, B \in M_{n\times n}[F]$, $f(A,B) = A \cdot B$. Assume that we have a program P such that $\mathrm{error}(f, P, \mathcal{U}_{M_{n\times n}[F]}) \le 1/16$. The following program is a 1/16-self-correcting program for f making calls to oracle P with respect to $\mathcal{U}_{M_{n\times n}[F]}$. The input to the program is $A, B \in M_{n\times n}[F]$ and the confidence parameter β.

Program Matrix Multiplication Self-Correct(A, B, β)

```
$N \leftarrow 12 \ln(1/\beta)$
Do for $m = 1, \ldots, N$
  Choose $A_1, B_1 \in_{\mathcal{U}} M_{n\times n}[F]$
  $A_2 \leftarrow A - A_1$
  $B_2 \leftarrow B - B_1$
  $answer_m \leftarrow P(A_1, B_1) + P(A_2, B_1) + P(A_1, B_2) + P(A_2, B_2)$
Output the most common answer among $\{answer_m : m = 1, \ldots, N\}$
```

Lemma 7 *The above program is a 1/16-self-correcting program for matrix multiplication.*

PROOF: Follows the outline of the proof of Lemma 2. ■

3.7 Polynomial Multiplication

We consider multiplication of polynomials over a ring. Let $R^d[x]$ denote the set of polynomials of degree d with coefficients from some ring R, and let $\mathcal{U}_{R^d[x]\times R^d[x]}$ be the uniform distribution on $R^d[x] \times R^d[x]$. In this case, $f(p(x), q(x)) = p(x) \cdot q(x)$, where $p, q \in R^d[x]$. Assume that we have a program P such that $\mathrm{error}(f, P, \mathcal{U}_{R^d[x]\times R^d[x]}) \le 1/16$. The following program is a 1/16-self-correcting program for f making oracle calls to P with respect to $\mathcal{U}_{R^d[x]\times R^d[x]}$. The input to the program is $p, q \in R^d[x]$ and the confidence parameter β.

Program Polynomial Multiplication Self-Correct(p, q, β)

```
$N \leftarrow 12 \ln(1/\beta)$
Do for $m = 1, \ldots, N$
  Choose $p_1 \in_{\mathcal{U}} R^d[x]$
  Choose $q_1 \in_{\mathcal{U}} R^d[x]$
  $p_2 \leftarrow p - p_1$
  $q_2 \leftarrow q - q_1$
  $answer_m \leftarrow P(p_1, q_1) + P(p_2, q_1) + P(p_1, q_2) + P(p_2, q_2)$
Output the most common answer among $\{answer_m : m = 1, \ldots, N\}$
```

Lemma 8 *The above program is a 1/16-self-correcting program for polynomial multiplication.*

PROOF: Follows the outline of the proof of Lemma 2. ■

3.8 Generic Self-Correcting Program

Let c be a positive integer and let f be any c-random self-reducible function (see page 7 for the definition). Assume that we have a program P such that $\text{error}(f, P, \mathcal{D}_n) \leq \frac{1}{4c}$. The following program is a $\frac{1}{4c}$-self-correcting program for f making oracle calls to P with respect to $\mathcal{D}_n$. The input to the program is n, $x \in \mathcal{I}_n$ and a confidence parameter β.

Program Generic Self-Correct(n, x, β)

```
$N \leftarrow 12 \ln(1/\beta)$
Do for $m = 1, \ldots, N$
  Randomly generate $a_1, \ldots, a_c$ based on $x$
  For $i = 1, \ldots, c$, $\alpha_i \leftarrow P(a_i)$
  $answer_m \leftarrow F(x, a_1, \ldots, a_c, \alpha_1, \ldots, \alpha_c)$
Output the most common answer among $\{answer_m : m = 1, \ldots, N\}$
```

Theorem 1 *If f is a function that is c-random self-reducible then there is a $\frac{1}{4c}$-self-correcting program for f.*

PROOF: We show that the above program is such a self-correcting program for f. Suppose that $\text{error}(f, P, \mathcal{D}_n) \leq \frac{1}{4c}$. Because, for each $k = 1, \ldots, c$, a_k is randomly distributed in $\mathcal{I}_n$ according to $\mathcal{D}_n$, all c outputs of P are correct with probability at least 3/4 each time through the loop. If all c outputs of P are correct, then by the random self-reducibility property, $answer_m = f(x)$. The theorem follows from Proposition 1. ■

4 Linearity and Self-Testing

Although the most interesting of our self-testing methods leads to self-testers that are almost as simple to code as the self-correctors described above, the proofs that they meet their specifications are more difficult, interesting and involve some probability theory on groups that may have other applications. This method applies to integer multiplication, the mod function, modular multiplication, modular exponentiation when the ϕ function of the modulus is known, and integer division. The resulting self-testers are simple to code, and are both different and efficient.

To give some idea of how the method works, we concentrate on the mod function. We then define the *linearity property*, and give a generic tester that works for any function with this property. We then show the specific testers that result from applying this generic tester to integer multiplication, modular multiplication, modular exponentiation and integer division.

4.1 Mod Function

For positive integers x and R, let $f(x, R) = x \bmod R$. Because the self-correcting program for the mod function relies on a program that is correct for most inputs with respect to a particular modulos R, the self-testing program for the mod function is designed to self-test with respect to a fixed modulus R. This is an important motivation for constructing efficient self-testing programs, because the self-testing program is executed each time a new modulus is used. Similar remarks hold for modular multiplication and modular exponentiation.

For fixed R, we view f as a function of one input x. There are two critical tests performed by the self-tester. Let $x_1 \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$ and $x_2 \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$ be independently chosen, and set $x \leftarrow x_1 +_{R2^n} x_2$. Note that $f(x, R) = f(x_1, R) +_R f(x_2, R)$, i.e. f is a (modular) linear function of its first input. The *linear consistency test* is

$$\text{“Does } P(x, R) = P(x_1, R) +_R P(x_2, R)\text{?”},$$

and the *linear consistency error* is the probability that the answer to the linear consistency test is “no”. Let $z \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$, and set $z' \leftarrow z +_{R2^n} 1$. Note that $f(z', R) = f(z, R) +_R 1$, i.e. in addition to being linear in its first input, f also has (modular) slope one as a function of its first input. The *neighbor consistency test* is

$$\text{“Does } P(z', R) = P(z, R) +_R 1\text{?”},$$

and the *neighbor consistency error* is the probability that the answer to the neighbor consistency test is “no”.

Our main theorem with respect to the self-tester for f is that there are constants $0 < \psi < 1$ and $\psi' > 1$ such that $\text{error}(f, P, \mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\{R\}})$ is at least ψ times the minimum of the linear consistency error and the neighbor consistency error, and that $\text{error}(f, P, \mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\{R\}})$ is at most ψ' times the maximum of the linear consistency error and the neighbor consistency error. Thus, we can *indirectly* approximate $\text{error}(f, P, \mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\{R\}})$ by instead estimating the linear and neighbor consistency errors.

The proof of the theorem shows that any function which usually satisfies the linearity property is essentially a linear function, in the sense that there is some linear function which is almost always equal to the original function.

Program Mod Function Self-Test(n, R, β)

```
    $N = 864 \ln(4/\beta)$
    $t \leftarrow 0$
    Do for $m = 1, \ldots, N$
      Call **Mod_Linear_Test** $(n, R, answer)$
      $t \leftarrow t + answer$
    If $t/N > 1/72$ then output “FAIL”

    $N' = 32 \ln(4/\beta)$
    $t' \leftarrow 0$
    Do for $m = 1, \ldots, N'$
      Call **Mod_Neighbor_Test** $(n, R, answer)$
      $t' \leftarrow t' + answer$
    If $t'/N' > 1/4$ then output “FAIL” else output “PASS”
```

Mod_Linear_Test $(n, R, answer)$

```
Choose $x_1 \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$
Choose $x_2 \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$
$x \leftarrow x_1 +_{R2^n} x_2$
If $P(x_1, R) +_R P(x_2, R) = P(x, R)$ then $answer \leftarrow 0$ else $answer \leftarrow 1$
```

Mod_Neighbor_Test $(n, R, answer)$:

```
Choose $z \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$
$z' \leftarrow z +_{R2^n} 1$
If $P(z, R) +_R 1 = P(z', R)$ then $answer \leftarrow 0$ else $answer \leftarrow 1$
```

Theorem 2 *The above program is an* $(1/432, 1/8)$*-self-testing program for the mod function with any modulus* R.

PROOF: This is a corollary of Theorem 6 from the next subsection. ■

The only non-trivial lines of code in the self-testing program are generation of random numbers, calls to the program P, integer additions and integer comparisons.

4.2 Generic Linear Self-Testing

In this section, we describe a generalization of the mod function self-tester to functions f mapping a group G into another group G'. In addition to the mod function, we will show how to apply this generic self-tester to integer multiplication, modular multiplication and modular exponentiation. In all cases, the resulting self-testing program is extremely simple to code, different and efficient.

The function f is specified in the following way. Let $(G, \circ)$ be a finite group with group elements G and group operation $\circ$ generated by the set $\{g_1, \ldots, g_c\}$ and with identity element 0. For $y \in G$, let y^{-1} denote the inverse of y. Let G' be a group with group operation $\circ'$ and identity element $0'$. For $\alpha \in G'$, let α^{-1} denote the inverse of α. Let $f : G \rightarrow G'$. Intuitively, f is hard to compute compared to either the $\circ$ or $\circ'$ operations. For simplicity, we assume that both groups are abelian; our results can be generalized to non-abelian groups as well, but our applications are to abelian groups.

If there are no finite subgroups of G' other than $\{0'\}$ then the rest of the characterization of f is solely in terms of a function F_{linear} as described below. If there are finite subgroups of G' other than $\{0'\}$ (including possibly G' itself) then, in addition to F_{linear}, the rest of the characterization is in terms of a function F^i_{neighbor} for each $i = 1, \ldots, c$, The fact that f is completely characterized by this information follows as a corollary from the theorems proved in the remainder of this subsection.

DEFINITION 4.1 (F_{linear} and linear consistency) *For any pair* $x_1, x_2 \in G$, $F_{\text{linear}}(x_1, x_2) \in G'$ *and furthermore* $f(x_1 \circ x_2) = f(x_1) \circ' f(x_2) \circ' F_{\text{linear}}(x_1, x_2)$. *We call this property* linear consistency.

In all of our applications except for integer multiplication, $F_{\text{linear}}(x_1, x_2) = 0'$ for all inputs x_1, x_2, in which case f is a group homomorphism.

DEFINITION 4.2 (F^i_{neighbor} and neighbor consistency) *For each generator $g_i \in G$, for any $z \in G$, $F^i_{\text{neighbor}}(z) \in G'$ and furthermore $f(z \circ g_i) = f(z) \circ' F^i_{\text{neighbor}}(z)$. We call this property* neighbor consistency.

The functions F^i_{neighbor} are not needed to characterize integer multiplication (because the group corresponding to G' is infinite with no finite subgroups other than $\{0'\}$ in that application). For all of the other applications, both G and G' are generated by a single element denoted 1 and $1'$, respectively, (i.e. they are both cyclic groups), and F^1_{neighbor} is a constant function.

The assumptions we are making is that the self-tester can easily compute the $\circ$ and $\circ'$ operations and the function F_{linear}. Furthermore, we assume that the self-tester can easily determine membership in G and G', and can easily choose a random element from G uniformly. In the case when G' has finite subgroups other than $\{0'\}$, for each $i = 1, \ldots, c$ we assume that g_i is easy to compute and that F^i_{neighbor} is easy to compute. The implicit assumption is that it is much harder to compute f directly then any one of these computations. In all of our applications, this is the case. We say a function f that is characterized as above has the *linearity property*.

The linearity property is a special case of 2-random self-reducibility. This can be seen as follows: Given x, choose $x_1 \in_{\mathcal{U}} G$ and let $x_2 \leftarrow x \circ x_1^{-1}$. Then, $f(x) = F_{\text{random}}(x, x_1, x_2, f(x_1), f(x_2))$, where $F_{\text{random}}(x, x_1, x_2, f(x_1), f(x_2))$ is defined to be $f(x_1) \circ' f(x_2) \circ' F_{\text{linear}}(x_1, x_2)$.

Let P be a program that supposedly computes f such that, for all $y \in G$, $P(y) \in G'$. **Generic Self-Test 1** is an $(\epsilon/54, \epsilon)$-self-tester for f with respect to $\mathcal{U}_G$ when G' has no finite subgroups other than $\{0'\}$. The self-tester for integer multiplication is based on **Generic Self-Test 1**, where $G = \mathcal{Z}_{2^n}$ with $+_{2^n}$ as the group operation, and $G' = \mathcal{Z}$ with $+$ as the group operation. The integer division self-tester is also based on **Generic Self-Test 1**. **Generic Self-Test 2** is an $(\epsilon/54, \epsilon)$-self-testing program for f with respect to $\mathcal{U}_G$ for all other G'. The self-tester for the mod function described in Subsection 4.1, for modular multiplication and for modular exponentiation are all based on **Generic Self-Test 2**. To avoid unnecessary complications in the description, as before (see page 7) we assume that whenever the program P is called that the self-tester checks to see if the answer returned is in G', and if it is not then the returned value is set to $0'$.

Program Generic Self-Test 1(ϵ, β)

```
(When $G'$ has no finite subgroups other than $\{0'\}$,
in which case there are no $F^i_{\text{neighbor}}$ functions specified.)
$N \leftarrow \frac{72}{\epsilon}\ln(2/\beta)$
$t \leftarrow 0$
Do for $m = 1, \ldots, N$
    Call **Generic_Linear_Test**$(answer)$
    $t \leftarrow t + answer$
If $t/N > \epsilon/9$ then output "FAIL" else output "PASS"
```

Program Generic Self-Test 2(ϵ, β)

```
(When $G'$ has finite subgroups (possibly $G'$ itself) other than $\{0'\}$,
in which case the $F^i_{\text{neighbor}}$ functions are specified.)
$N \leftarrow \frac{72}{\epsilon}\ln(4/\beta)$
```

```
$t \leftarrow 0$
Do for $m = 1, \ldots, N$
  Call **Generic_Linear_Test**($answer$)
  $t \leftarrow t + answer$
If $t/N > \epsilon/9$ then output "FAIL"

$N' \leftarrow 32 \ln(4c/\beta)$
$t' \leftarrow 0$
Do for $m = 1, \ldots, N'$
  $answer \leftarrow 0$
  For $i = 1, \ldots, c$, call **Generic_Neighbor_Test**($i, answer$)
  $t' \leftarrow t' + answer$
If $t'/N' > 1/4$ then output "FAIL" else output "PASS"
```

Generic_Linear_Test ($answer$)

```
Choose $x_1 \in_{\mathcal{U}} G$.
Choose $x_2 \in_{\mathcal{U}} G$.
If $P(x_1 \circ x_2) = P(x_1) \circ' P(x_2) \circ' F_{\text{linear}}(x_1, x_2)$ then $answer \leftarrow 0$ else $answer \leftarrow 1$
```

Generic_Neighbor_Test ($i, answer$)

```
Choose $z \in_{\mathcal{U}} G$.
If $P(z \circ g_i) \neq P(z) \circ' F^i_{\text{neighbor}}(z)$ then $answer \leftarrow 1$
```

Before giving proofs, we first introduce some notation and provide intuition for why the self-testers work. For each $y \in G$, define the *discrepancy of y* to be

$$\text{disc}(y) \equiv f(y) \circ' P(y)^{-1}.$$

Note that P computes f correctly for all inputs if and only if the discrepancy function defines a homomorphism from G into $\{0'\}$.

Because of linear consistency and because the self-testing program computes $F_{\text{linear}}(x_1, x_2)$ correctly on its own, $P(x_1 \circ x_2) = P(x_1) \circ' P(x_2) \circ' F_{\text{linear}}(x_1, x_2)$ if and only if

$$\text{disc}(x_1 \circ x_2) = \text{disc}(x_1) \circ' \text{disc}(x_2).$$

If this equality holds for all $x_1, x_2 \in G$ then the discrepancy function defines a homomorphism h from G into G'. Intuitively, **Generic_Linear_Test** verifies that the discrepancy function is "close" to some homomorphism h.

Suppose that G' has no finite subgroup other than $\{0'\}$. Then, because G is finite, h is the trivial mapping from G to $\{0'\}$. Now suppose G' has a finite subgroup other than $\{0'\}$. Because of neighbor consistency and because the self-testing program computes $F^i_{\text{neighbor}}(z)$ correctly on its own, $P(z \circ g_i) = P(z) \circ' F^i_{\text{neighbor}}(z)$ if and only if

$$\text{disc}(z \circ g_i) = \text{disc}(z).$$

If, for all $z \in G$ and for all $i = 1, \ldots, c$, $\mathrm{disc}(z \circ g_i) = \mathrm{disc}(z)$ then h is the trivial mapping from G to $\{0'\}$, and **Generic_Neighbor_Test** is used to verify this.

The following notation is used throughout the rest of this section.

Notation:

- $\delta \equiv \Pr[\mathrm{disc}(x_1 \circ x_2) \neq \mathrm{disc}(x_1) \circ' \mathrm{disc}(x_2)]$ when $x_1 \in_{\mathcal{U}} G$ and $x_2 \in_{\mathcal{U}} G$ are independently chosen.
- For all $i = 1, \ldots, c$, $\delta_i \equiv \Pr[\mathrm{disc}(z) \neq \mathrm{disc}(z \circ g_i)]$ when $z \in_{\mathcal{U}} G$.
- $\psi \equiv \Pr[\mathrm{disc}(y) \neq 0']$ when $y \in_{\mathcal{U}} G$.

Theorems 3 and 4 are the heart of the proof that programs **Generic Self-Test 1** and **Generic Self-Test 2** meet their specifications, respectively.

Theorem 3 *Let G' be a group with no finite subgroups except for $\{0'\}$. Then, $\delta \geq 2\psi/9$.*

Theorem 4 *Let G' be any group. If, for all $i = 1, \cdots, c$, $\delta_i < 1/2$, then $\delta \geq 2\psi/9$.*

The specific proofs we give of Theorems 3 and 4, due largely to Don Coppersmith, are simpler than our original proofs. A full exposition of some related general probability results will appear in [8, Ben-Or Coppersmith Luby Rubinfeld]. We now introduce some more notation and prove some intermediate lemmas that are used in the proofs of Theorems 3 and 4.

Uncapitalized letters from the end of the alphabet denote elements chosen randomly from G according to $\mathcal{U}_G$, e.g. x, y and z, whereas uncapitalized letters from the beginning of the alphabet denote fixed elements of G, e.g. a, b, c. For Lemmas 9, 10, 11 and 12, we assume that $\delta < 2/9$. Let δ' be defined as the solution to the equality $\delta'(1-\delta') = \delta$. Because $\delta < 2/9$, $\delta' < 1/3$.

Lemma 9 *$\forall a \in G$, $\exists a' \in G'$ such that $\Pr[\mathrm{disc}(x \circ a) = \mathrm{disc}(x) \circ' a'] \geq 1 - \delta'$.*

PROOF: By the definition of δ and because $x \circ a$ is distributed in G according to $\mathcal{U}_G$ and $a \circ y$ is distributed in G according to $\mathcal{U}_G$,

$$\Pr[\mathrm{disc}(x \circ a) \circ' \mathrm{disc}(y) = \mathrm{disc}(x \circ a \circ y)$$
$$= \mathrm{disc}(x) \circ' \mathrm{disc}(a \circ y)] \geq 1 - 2\delta.$$

So

$$\Pr[\mathrm{disc}(x \circ a) \circ' \mathrm{disc}(x)^{-1} = \mathrm{disc}(y \circ a) \circ' \mathrm{disc}(y)^{-1}]$$
$$\geq 1 - 2\delta.$$

This is the sum, over all $a' \in G'$, of the square of the probability

$$\Pr[\mathrm{disc}(x \circ a) \circ' \mathrm{disc}(x)^{-1} = a'].$$

Since $\delta < 2/9$, this sum exceeds $5/9$ and thus there must be one value a' with

$$\Pr[\mathrm{disc}(x \circ a) \circ' \mathrm{disc}(x)^{-1} = a'] \geq 1 - \delta'$$

where $(1-\delta')^2 + \delta'^2 = 1 - 2\delta$ and $\delta' < 1/2$. This leads to $\delta'(1-\delta') = \delta$. ■

DEFINITION 4.3 (the function h) *Lemma 9 leads to the definition of the function h from G to G' defined as follows: For all $a \in G$, let $h(a) = a'$, where a' is the element of G' described in Lemma 9.*

Lemma 10 *The function h is a group homomorphism from G to G', i.e. for all $a, b \in G$, $h(a \circ b) = h(a) \circ' h(b)$.*

PROOF: Using Lemma 9 three times, for all $a, b \in G$,

$$\Pr[\mathrm{disc}(x) \circ' h(a) \circ' h(b) = \mathrm{disc}(x \circ a) \circ' h(b) =$$

$$\mathrm{disc}(x \circ a \circ b) = \mathrm{disc}(x) \circ' h(a \circ b)] \geq 1 - 3\delta'.$$

This probability is strictly greater than zero because $\delta' < 1/3$, and thus $h(a \circ b) = h(a) \circ' h(b)$. ■

Lemma 11

(1) *If G' is a group with no finite subgroups except for $\{0'\}$ then for all $a \in G$, $h(a) = 0'$.*

(2) *If G' is any group and, for all $i = 1, \cdots, c$, $\delta_i < 1/2$, then for all $a \in G$, $h(a) = 0'$.*

PROOF: By Lemma 10, h is a group homomorphism and thus the image of h is a finite subgroup of G'. In case (1), the only finite subgroup of G' is $\{0'\}$. In case (2), consider a fixed $i \in \{1, \ldots, c\}$. Because $1 - \delta_i > 1/2$ and using Lemma 9 and the fact that $1 - \delta' > 2/3$,

$$\Pr[\mathrm{disc}(x) = \mathrm{disc}(x \circ g_i) = \mathrm{disc}(x) \circ' h(g_i)] > 1/6,$$

and thus there is some $x \in G$ such that $\mathrm{disc}(x) = \mathrm{disc}(x) \circ' h(g_i)$ which implies that $h(g_i) = 0'$. Thus, for all $i = 1, \ldots, c$, $h(g_i) = 0'$. Because $g_1, \ldots, g_c$ are generators for G it follows that for all $a \in G$, $h(a) = 0'$. ■

Lemma 12 *Under the same conditions as (1) and (2) in Lemma 11,* $\Pr[\mathrm{disc}(x) = \mathrm{disc}(x \circ y)] \geq 1 - \delta'$.

PROOF: By Lemma 11, $h(a) = 0'$ for all $a \in G$. On the other hand, Lemma 9 says that

$$\Pr[\mathrm{disc}(x \circ a) = \mathrm{disc}(x) \circ' h(a)] \geq 1 - \delta'$$

for every $a \in G$, and thus certainly this is true when a is replaced with a random y. Thus, $\Pr[\mathrm{disc}(x \circ y) = \mathrm{disc}(x)] \geq 1 - \delta'$. ■

PROOF: [of Theorem 3] Assume first that $\delta < 2/9$. By definition of δ and using Lemma 12, $\Pr[\mathrm{disc}(x) = \mathrm{disc}(x \circ y) = \mathrm{disc}(x) \circ' \mathrm{disc}(y)] \geq 1 - \delta' - \delta$, and thus $\Pr[\mathrm{disc}(y) = 0'] \geq 1 - \delta' - \delta$ which implies that $\psi \leq \delta + \delta'$. Because $\delta' < 1/3$, $1 - \delta' > 2/3$ which implies that $\delta' \leq 3\delta/2$. This implies that $\delta \geq 2\psi/5$. On the other hand, if $\delta \geq 2/9$, then because $\psi \leq 1$ it follows that $\delta \geq 2\psi/9$. ■

PROOF: [of Theorem 4] Analogous to the proof of Theorem 3. ■

Theorems 3 and 4 provide the upper bounds on ψ in terms of δ and $\delta_1, \ldots, \delta_c$. We now develop the easier to prove lower bounds on ψ.

Lemma 13 *Let G' be any group. Then, $3\psi \geq \delta$.*

PROOF: Because $1-\psi = \Pr[\mathrm{disc}(y) = 0']$, $\Pr[\mathrm{disc}(x_1 \circ x_2) = \mathrm{disc}(x_1) = \mathrm{disc}(x_2) = 0'] \geq 1-3\psi$, and consequently $\delta = \Pr[\mathrm{disc}(x_1 \circ x_2) \neq \mathrm{disc}(x_1) \circ' \mathrm{disc}(x_2)] \leq 3\psi$. ■

Lemma 14 *Let G' be any group. Then, for all $i = 1, \ldots, c$, $\psi \geq \delta_i/2$.*

PROOF: For all $i = 1, \ldots, c$, if $\mathrm{disc}(z \circ g_i) \neq \mathrm{disc}(z)$ then either $\mathrm{disc}(z \circ g_i) \neq 0'$ or $\mathrm{disc}(z) \neq 0'$. Thus, $\psi \geq \delta_i/2$. ■

The following proposition is used to quantify the number of random samples needed to guarantee good estimates of δ and $\delta_1, \ldots, \delta_c$ with high probability. This proposition can be proved using standard techniques from an inequality due to Bernstein cited in [29, Rènyi]. For a proof of this proposition, see for example [25, Karp Luby Madras].

Proposition 15 *Let $Y_1, Y_2, \ldots$ be independent identically distributed 0/1-valued random variables with mean μ. Let $\theta \leq 2$. If $N \geq \frac{1}{\mu} \cdot \frac{4\ln(2/\beta)}{\theta^2}$ then $\Pr[(1-\theta)\mu \leq \tilde{Y} \leq (1+\theta)\mu] \geq 1-\beta$, where $\tilde{Y} = \sum_{i=1}^{N} Y_i/N$.*

Corollary 16 *Let $Y_1, Y_2, \ldots$ be independently distributed 0/1-valued random variables with means $\mu_1, \mu_2, \ldots$, respectively.*

(1) *If, for all i, $\mu_i \geq \mu$ and $N = \frac{1}{\mu} \cdot 16\ln(2/\beta)$ then $\Pr[\tilde{Y} \leq \mu/2] \leq \beta$, where $\tilde{Y} = \sum_{i=1}^{N} Y_i/N$. (Use $\theta = 1/2$.)*

(2) *If, for all i, $\mu_i \leq \mu$ and $N = \frac{1}{\mu} \cdot 4\ln(2/\beta)$ then $\Pr[\tilde{Y} \geq 2\mu] \leq \beta$, where $\tilde{Y} = \sum_{i=1}^{N} Y_i/N$. (Use $\theta = 1$.)*

Theorem 5 *Let f be a function as specified above in the case when G' has no finite subgroups other than $\{0'\}$. Then, for any input parameter $0 \leq \epsilon \leq 1$,* **Generic Self-Test 1** *is an $(\epsilon/54, \epsilon)$-self-tester for f.*

PROOF:

($\psi \geq \epsilon$) By Theorem 3, this implies that $\delta \geq 2\epsilon/9$. Letting $\mu = 2\epsilon/9$ and letting $N = \frac{1}{\mu} \cdot 16\ln(2/\beta) = \frac{72}{\epsilon}\ln(2/\beta)$ and using the Corollary 16, Part (1) yields $\Pr[total/N \leq \epsilon/9] \leq \beta$. On the other hand, if $total/N > \epsilon/9$ then the output of the program is "FAIL". Thus, if $\psi \geq \epsilon$, the program outputs "FAIL" with probability at least $1-\beta$.

($\psi \leq \epsilon/54$) Lemma 13 implies that $\delta \leq \epsilon/18$. Letting $\mu = \epsilon/18$ and letting $N = \frac{1}{\mu} \cdot 4\ln(2/\beta) = \frac{72}{\epsilon}\ln(2/\beta)$ and using the Corollary 16, Part (2), yields $\Pr[total/N \geq \epsilon/9] \leq \beta$. On the other hand, if $total/N < \epsilon/9$ then the output of the program is "PASS". Thus, if $\psi \leq \epsilon/54$, the program outputs "PASS" with probability at least $1-\beta$.

■

Theorem 6 *Let f be a function as specified above in the case when G' has finite subgroups other than $\{0'\}$. Then, for any input parameter $0 \le \epsilon \le 1$,* **Generic Self-Test 2** *is an $(\epsilon/54, \epsilon)$-self-tester for f.*

PROOF:

($\psi \ge \epsilon$) We partition the possibilities into two subcases: (1) For all $i = 1, \ldots, c$, $\delta_i < 1/2$; (2) There is an $i = 1, \ldots, c$ such that $\delta_i \ge 1/2$. Case (1) is similar to the $\psi \ge \epsilon$ case of Theorem 5, using Theorem 4 in place of Theorem 3, which yields that the program outputs "FAIL" with probability at least $1 - \beta/2$. In case (2), because of the Corollary 16, Part (1), letting $\mu = 1/2$ and letting $N = 32 \ln(4c/\beta)$ yields $\Pr[total'/N' \le 1/4] \le \frac{\beta}{2c}$. On the other hand, if $total'/N' > 1/4$ then the output of the program is "FAIL", and thus the program outputs "FAIL" with probability at least $1 - \frac{\beta}{2c}$. Thus, in either case, the program outputs "FAIL" with probability at least $1 - \beta$.

($\psi \le \epsilon/54$) We partition the possibilities into two subcases: (1) For all $i = 1, \ldots, c$, $\delta_i \le 1/8$; (2) There is an $i = 1, \ldots, c$ such that $\delta_i > 1/8$. A portion of case (1) is similar to the $\psi \le \epsilon/54$ case of Theorem 5, which yields $\Pr[total/N > \epsilon/9] \le \beta/2$. Also in case (1), using the Corollary 16, Part (2), letting $\mu = 1/8$ and letting $N = 32\ln(4c/\beta)$ and, using the fact that the union of c probabilities is upper bounded by their sum, yields $\Pr[total'/N' > 1/4] \le \beta/2$. Thus, in case (1) the program outputs "PASS" with probability at least $1 - \beta$. In case (2), because of Lemma 14, there is some i such that $\delta_i > 1/8$ implies that $\psi > 1/16 > \epsilon/54$ since $\epsilon \le 1$. Thus case (2) is impossible.

■

4.3 Integer Multiplication

For positive integers x and y, let $f(x, y) = x \cdot y$. We now describe in what sense integer multiplication has the linearity property. For any triple of integers x_1, x_2 and y, $x_1 \cdot y + x_2 \cdot y = (x_1 + x_2) \cdot y$. Thus, for a fixed value of y, integer multiplication is a linear function. For the following discussion, fix y to an arbitrary value. In this case, f can be viewed of as a function of one input with domain $G = \mathcal{Z}_{2^n}$ where $\circ$ is $+_{2^n}$, and range $G' = \mathcal{Z}$ where $\circ'$ is $+$. For $x_1, x_2 \in \mathcal{Z}_{2^n}$, let $c = 1$ if $x_1 + x_2 \ge 2^n$ and let $c = 0$ otherwise, and let $x = x_1 + x_2 - c2^n = x_1 +_{2^n} x_2$. At the heart of the integer multiplication self-testing program is the fact that $f(x_1, y) + f(x_2, y) = f(x, y) + yc2^n$. Note that $F_{\mathrm{linear}}(x_1, x_2) = yc2^n$ is easily computable.

Based on **Generic Self-Test 1** with $\epsilon = 1/16$, the following program is an $(1/864, 1/16)$-self-testing program for f making oracle calls to P with respect to $\mathcal{U}_{\mathcal{Z}_{2^n}} \times \mathcal{U}_{\mathcal{Z}_{2^n}}$. The input to the program is n and the confidence parameter β.

Program Integer Multiplication Self-Test(n, β)

```
    $N = 1152 \ln(2/\beta)$
    $total \leftarrow 0$
    Do for $m = 1, \ldots, N$
      Call **Int_Mult_Linear_Consistency**$(n, answer)$
      $total \leftarrow total + answer$
```

```
If $total/N > 1/144$ then output "FAIL" else output "PASS"
```

Int_Mult_Linear_Consistency$(n, answer)$

```
Choose $y \in_{\mathcal{U}} \mathcal{Z}_{2^n}$
Choose $x_1 \in_{\mathcal{U}} \mathcal{Z}_{2^n}$
Choose $x_2 \in_{\mathcal{U}} \mathcal{Z}_{2^n}$
$x \leftarrow x_1 +_{2^n} x_2$
$c \leftarrow (x_1 + x_2)$ div $2^n$
If $P(x_1, y) + P(x_2, y) = P(x, y) + cy2^n$ then $answer \leftarrow 0$ else $answer \leftarrow 1$
```

Theorem 7 *The above program is a* $(1/864, 1/16)$*-self-testing program for integer multiplication.*

PROOF: Similar to the proof of Theorem 5, except that for each y there is a different value for $\psi(y)$ and ψ is the average of $\psi(y)$ over all y. For the first part of the proof, note that $\delta(y) \geq 2\psi(y)/9$ for each value of y. Thus, if $\psi = E[\psi(y)] \geq \epsilon$ then $\delta = E[\delta(y)] \geq 2\epsilon/9$. The rest of the proof is the same for case 1. Similar comments hold for the second case of the proof. ■

The integer multiplication self-testing program is both different and efficient. The only non-trivial lines of code in the self-testing program are generation of random numbers, calls to the program P, integer additions, shifts and integer comparisons.

4.4 Modular Multiplication

For positive integers x, y and R, let $f(x, y, R) = x \cdot_R y$. For fixed value for R and y, f can be thought of as a function of x. In this case, the domain of f can be thought of as $G = \mathcal{Z}_{R2^n}$ where $\circ$ is $+_{R2^n}$ and the range of f is $G' = \mathcal{Z}_R$ where $\circ'$ is $+_R$. The heart of the modular multiplication self-testing program is the fact that, for any pair $x_1, x_2 \in \mathcal{Z}_{R2^n}$, $f(x_1, y, R) +_R f(x_2, y, R) = f((x_1 +_{R2^n} x_2), y, R)$. Thus, $F_{\text{linear}}(x_1, x_2) = 0'$. The generator for G is 1. It is easy to see that $f((z +_{R2^n} 1), y, R) = f(z, y, R) +_R y$, and thus $F^1_{\text{neighbor}}(z) = y$.

Based on **Generic Self-Test 2** with $\epsilon = 1/16$, the following program is an $(1/864, 1/16)$-self-testing program for f with respect to $\mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\{R\}}$. The input to the program is n, R and the confidence parameter β.

Program Modular Multiplication Self-Test(n, R, β):

```
$N = 1152 \ln(4/\beta)$
$total \leftarrow 0$
Do for $m = 1, \ldots, N$
  Call Mult_Mod_Linear_Consistency$(n, R, answer)$
  $total \leftarrow total + answer$
If $total/N > 1/144$ then output "FAIL"

$N' = 32 \ln(4/\beta)$
$total' \leftarrow 0$
Do for $m = 1, \ldots, N'$
  Call Mult_Mod_Neighbor_Consistency$(n, R, answer)$
```

```
    $total' \leftarrow total' + answer$
  If $total'/N' > 1/4$ then output "FAIL" else output "PASS"
```

Mult_Mod_Linear_Consistency$(n, R, answer)$

```
  Choose $y \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$
  Choose $x_1 \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$
  Choose $x_2 \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$
  $x \leftarrow x_1 +_{R2^n} x_2$
  If $P(x_1, y, R) +_R P(x_2, y, R) = P(x, y, R)$ then $answer \leftarrow 0$ else $answer \leftarrow 1$
```

Mult_Mod_Neighbor_Consistency$(n, R, answer)$:

```
  Choose $y \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$
  Choose $z \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$
  $z' \leftarrow z +_{R2^n} 1$
  If $P(z, y, R) +_R y = P(z', y, R)$ then $answer \leftarrow 0$ else $answer \leftarrow 1$
```

Theorem 8 *The above program is an* $(1/864, 1/16)$*-self-testing program for modular multiplication.*

PROOF: See the proof of Theorem 6, and combine this with some of the aspects of the proof of Theorem 7. ■

The only non-trivial lines of code in the self-testing program are generation of random numbers, calls to the program P, integer additions and integer comparisons, except for the line "If $P(z, y, R) +_R y = P(z', y, R)$ then ... " in the **Mult_Mod_Neighbor_Consistency** program. The problem is that although $P(z, y, R)$ and $P(z', y, R)$ are both in $\mathcal{Z}_R$, y is in the much larger range $\mathcal{Z}_{R2^n}$ and thus $y \bmod R$ cannot be calculated easily using just additions and comparisons.

This suggests using the library approach to get around this problem, i.e. use a library of functions including modular multiplication and the mod function. We have already presented a self-testing/correcting pair (T', C') for the mod R function. The modular multiplication self-testing program can then call C' to compute $y \bmod R$. C' computes this correctly with high confidence using any program P' for the mod R function that passes the test T'. Note that any modular multiplication program has the mod R function embedded in it, when restricting the inputs to multiplication by 1. The resulting modular multiplication self-testing program is both different and efficient.

4.5 Modular Exponentiation

For positive integers x, a and R, let $f(a, x, R) = a^x \bmod R$. Fix a and R to be positive integers, and as before we restrict attention to a and R such that $\gcd(a, R) = 1$ and we assume that we know the factorization of R and thus can easily compute $\phi(R)$. In this case, the domain of f is $G = \mathcal{Z}_{\phi(R)2^n}$ where $\circ$ is $+_{\phi(R)2^n}$ and the range of f is $G' = \mathcal{Z}_R^*$ and $\circ'$ is $\cdot_R$. Because $\gcd(a, R) = 1$, $a^{\phi(R)} = 1 \bmod R$. The heart of the modular exponentiation self-testing program is the fact that, for any pair $x_1, x_2 \in \mathcal{Z}_{\phi(R)2^n}$, $f(a, x_1, R) \cdot_R f(a, x_2, R) = f(a, x_1 +_{\phi(R)2^n} x_2, R)$. Thus, $F_{\text{linear}}(x_1, x_2) = 0'$.

The generator for G is 1. It is easy to see that $f(a, (z +_{\phi(R)2^n} 1), R) = f(a, z, R) \cdot_R a$, and thus $F^1_{\text{neighbor}}(z) = a$.

Based on **Generic Self-Test 2** with $\epsilon = 1/16$, the following program is an $(1/864, 1/16)$-self-testing program for f making oracle calls to P with respect to $\mathcal{U}_{\{a\}} \times \mathcal{U}_{\mathcal{Z}_{\phi(R)2^n}} \times \mathcal{U}_{\{R\}}$. The input to the program is n, a, R and the confidence parameter β.

Program Modular Exponentiation Self-Test(n, a, R, β)

```
$N = 1152 \ln(4/\beta)$
$total \leftarrow 0$
Do for $m = 1, \ldots, N$
  Call **Mod_Exp_Linear_Consistency**$(n, a, R, answer)$
  $total \leftarrow total + answer$
If $total/N > 1/144$ then output "FAIL"

$N' = 32 \ln(4/\beta)$
$total' \leftarrow 0$
Do for $m = 1, \ldots, N'$
  Call **Mod_Exp_Neighbor_Consistency**$(n, a, R, answer)$
  $total' \leftarrow total' + answer$
If $total'/N' > 1/4$ then output "FAIL" else output "PASS"
```

Mod_Exp_Linear_Consistency$(n, a, R, answer)$

```
Choose $x_1 \in_{\mathcal{U}} \mathcal{Z}_{\phi(R)2^n}$
Choose $x_2 \in_{\mathcal{U}} \mathcal{Z}_{\phi(R)2^n}$
$x \leftarrow x_1 +_{\phi(R)2^n} x_2$
If $P(a, x_1, R) \cdot_R P(a, x_1, R) = P(a, x, R)$ then $answer \leftarrow 0$ else $answer \leftarrow 1$
```

Mod_Exp_Neighbor_Consistency$(n, a, R, answer)$:

```
Choose $z \in_{\mathcal{U}} \mathcal{Z}_{\phi(R)2^n}$
$z' \leftarrow z +_{\phi(R)2^n} 1$
If $P(a, z, R) \cdot_R a = P(a, z', R)$ then $answer \leftarrow 0$ else $answer \leftarrow 1$
```

Theorem 9 *The above program is an* $(1/864, 1/16)$-*self-testing program for modular exponentiation.*

PROOF: Analogous to the proof of Theorem 8 (page 23). ■

The modular exponentiation self-testing program consists solely of integer additions, integer comparisons and calls to P except in two lines of code: (1) The line "If $P(a, x_1, R) \cdot_R P(a, x_1, R) = P(a, x, R)$ ..." in the program **Mod_Exp_Linear_Consistency**; (2) The line "If $P(a, z, R) \cdot_R a = P(a, z', R)$ ..." in the program **Mod_Exp_Neighbor_Consistency**. We propose computing these two lines using the library approach. We can use the modular multiplication self-correcting program presented above to compute (1) and (2) which uses a program P' for computing multiplication mod$R$, where we first use the modular multiplication self-testing program to verify that P'

is not too faulty. In addition to these two lines of code, in the implicit range-check code (see page 7) we need to verify that the answer α to a call to P is in range, i.e. in $\mathcal{Z}_R^*$. This can be done by verifying that $\alpha \in \mathcal{Z}_R$ (this is easy) and that $\gcd(\alpha, R) = 1$. If R is a prime, the gcd computation is trivial (just verify that $\alpha \neq 0$). If the prime factorization of R is $\prod_{i=1}^{\gamma} p_i^{e_i}$ where γ is a small positive integer then to verify that $\gcd(\alpha, R) = 1$, we can use the mod function self-correcting program to compute $\alpha \bmod p_i$ for all $i = 1, \ldots, \gamma$ and verify that none of the answers are zero. This requires that the mod function is not too faulty for modp_i computations for all $i = 1, \ldots, \gamma$. In subsection 7.4 we show how to reduce this requirement to the case where the mod function is not too faulty for mod$R$ computations. In this same subsection we present a self-testing/correcting pair for modular exponentiation when the prime factorization of R and $\phi(R)$ are not known, at the expense of some loss in efficiency.

4.6 Integer Division

We now consider division of integers by R for a positive number R. in this case, $f(x, R) = (x \text{ div } R, x \bmod R)$. We write $f_{div}(x, R) = x \text{ div } R$ and $f_{mod}(x, R) = x \bmod R$. We have already seen that the mod function has the linearity property. We now describe in what sense integer division has the linearity property. For any triple of integers x_1, x_2 and R, $x_1 \text{ div } R + x_2 \text{ div } R + (x_1 \bmod R + x_2 \bmod R) \text{ div } R = (x_1 + x_2) \text{ div } R$ and $x_1 \bmod R +_R x_2 \bmod R = x_1 +_R x_2$. For the following discussion, fix R to an arbitrary positive integer. In this case, f can be viewed of as a function of one input with domain $G = \mathcal{Z}_{R2^n}$ where $\circ$ is $+_{R2^n}$. The range G' of f is isomorphic to $\mathcal{Z}$, where $\circ'$ corresponds to $+$. An element of G' is a pair of integers (a, b), where $a \in \mathcal{Z}$ and $b \in \mathcal{Z}_R$. For any pair of elements $(a, b), (c, d) \in G'$, $(a, b) \circ' (c, d) = (a + c + (b + d) \text{ div } R, b +_R d)$. For $x_1, x_2 \in \mathcal{Z}_{R2^n}$, let $c = (x_1 + x_2) \text{ div } R2^n$ and let $x = x_1 + x_2 - cR2^n = x_1 +_{R2^n} x_2$. At the heart of the integer division self-testing program is the fact that $f_{div}(x, R) + c2^n = f_{div}(x_1, R) + f_{div}(x_2, R) + (f_{mod}(x_1, R) + f_{mod}(x_2, R)) \text{ div } R$ and that $f_{mod}(x, R) = f_{mod}(x_1, R) +_R f_{mod}(x_2, R)$

Based on **Generic Self-Test 1** with $\epsilon = 1/16$, the following program is an $(1/864, 1/16)$-self-testing program for f with respect to $\mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\{R\}}$. The input to the program is n, R and the confidence parameter β. We refer to the output of P as $P(x, R) = (P_{div}(x, R), P_{mod}(x, R))$.

Program Integer Division Self-Test(n, R, β)

```
    $N = 1152 \ln(2/\beta)$
    $total \leftarrow 0$
    Do for $m = 1, \ldots, N$
      Call **Int_Div_Linear_Consistency**$(n, R, answer)$
      $total \leftarrow total + answer$
    If $total/N > 1/144$ then output "FAIL" else output "PASS"
```

Int_Div_Linear_Consistency$(n, R, answer)$

```
    Choose $x_1 \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$
    Choose $x_2 \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$
    $x \leftarrow x_1 +_{R2^n} x_2$
    $c \leftarrow (x_1 + x_2) \text{ div } R2^n$
    $answer \leftarrow 0$
```

```
If $P_{div}(x_1, R) + P_{div}(x_2, R) + (P_{mod}(x_1, R) + P_{mod}(x_2, R))$ div $R \neq P_{div}(x, R) + c2^n$
    then $answer \leftarrow 1$
If $P_{mod}(x_1, R) +_R P_{mod}(x_2, R) \neq P_{mod}(x, R)$ then $answer \leftarrow 1$
```

Theorem 10 *The above program is a* $(1/864, 1/16)$*-self-testing program for integer division.*

PROOF: Similar to the proof of Theorem 5 (page 20). ■

5 Libraries

Often programs for related functions are grouped in packages; common examples include packages that solve statistics problems or packages that do matrix manipulations. It is reasonable therefore to use programs in these packages to help test and correct each other. We extend the theory proposed in [10, Blum] to allow the use of several programs, or a *library,* to aid in testing and correcting. We show that this allows one to construct self-testing/correcting pairs for functions which did not previously have efficient self-testing or self-correcting programs, or even result checkers. Thus, the self-testing/correcting pair is given a collection of programs, all of which are possibly faulty, and may call any one of them in order to test or correct a particular program. Working with a library of programs rather than with just a single program is a key idea: enormous difficulties arise in attempts to result check a determinant or rank program in the absence of programs for matrix multiplication and inverse.

The notion of libraries is useful for another reason as well: Consider again the problem of designing a self-testing/correcting pair for the determinant. Many of the proposed solutions require matrix multiplication. However, matrix multiplication and determinant are equivalent problems with respect to asymptotic running times [2, Aho Hopcroft Ullman]. Therefore, a determinant self-testing/correcting pair using matrix multiplication will not be quantifiably different from a program for the determinant. On the other hand, since matrix multiplication can be self-tested/corrected, one should not consider the complexity of the matrix multiplication routine towards the complexity of the self-testing/correcting pair for the determinant. In other words, the complexity of the self-testing/correcting pair should be evaluated as the complexity of the *unchecked parts* of the self-testing/correcting pair. The notion of libraries gives us a clean way of evaluating the complexity of the unchecked parts of the self-testing/correcting pair.

As an example of self-testing/correcting pairs written for a library of programs, we show how to self-test/correct a library of possibly fallible programs for matrix multiplication, matrix inverse, determinant and rank. As we informally discussed before, a library of self-testing/correcting pairs based on similar principles can be constructed for the following functions: integer mod, modular multiplication, modular exponentiation, and multiplicative inverse mod R. With such a library, the self-testing/correcting for all functions can be done with only a small number of additions, subtractions, comparisons and generation of random numbers.

5.1 Definitions

We give the following definitions, which generalize the previously given self-testing/correcting definitions.

Definition 5.1 (library) *A* library *is a family of functions $f^1, \ldots, f^c$ for some positive constant c. An* input set *for a library is a family $(\mathcal{I}^1, \mathcal{D}^1), \ldots, (\mathcal{I}^c, \mathcal{D}^c)$, where $\mathcal{D}^i_n$ is a distribution on inputs $\mathcal{I}^i_n$ to f^i. An* error set *for a library is a family of constants $\epsilon^1, \ldots, \epsilon^c$, where $0 < \epsilon^i < 1$.*

Definition 5.2 (library self-testing) *A* self-testing program *for a library $f^1, \ldots, f^c$ with input set $(\mathcal{I}^1, \mathcal{D}^1), \ldots, (\mathcal{I}^c, \mathcal{D}^c)$, error set $\epsilon^1_1, \ldots, \epsilon^c_1$ and error set $\epsilon^1_2, \ldots, \epsilon^c_2$, where, for each $i = 1, \ldots, c$, $\epsilon^i_1 < \epsilon^i_2$, is a probabilistic program T that has input n and β and makes calls to $P^1, \ldots, P^c$, where P^i supposedly computes f^i. T has the following properties:*

1. *If, for all $i = 1, \ldots, c$, $\text{error}(f^i, P^i, \mathcal{D}^i_n) \le \epsilon^i_1$ then T outputs "PASS" with probability at least $1 - \beta$.*

2. *If, for some $i = 1, \ldots, c$, $\text{error}(f^i, P^i, \mathcal{D}^i_n) \ge \epsilon^i_2$ then T outputs "FAIL" with probability at least $1 - \beta$.*

Definition 5.3 (library self-correcting) *A* self-correcting program *for f^1 with respect to a library $f^1, \ldots, f^c$ with input set $(\mathcal{I}^1, \mathcal{D}^1), \ldots, (\mathcal{I}^c, \mathcal{D}^c)$ and error set $\epsilon^1, \ldots, \epsilon^c$ is a probabilistic program C that on input n, $x \in \mathcal{I}^1_n$ and β makes calls to $P^1, \ldots, P^c$ to compute $C(x)$. C has the property that if, for all $i = 1, \ldots, c$, $\text{error}(f^i, P^i, D^i_n) \le \epsilon^i$ then, for all $x \in \mathcal{I}^1_n$, $C(x) = f^1(x)$ with probability at least $1 - \beta$.*

Definition 5.4 (library self-testing/correcting pair) *A* self-testing/correcting pair *for f^1 with respect to a library $f^1, \ldots, f^c$ is a pair of probabilistic programs (T, C) with the following properties. T is a self-testing program for the library with some input set $(\mathcal{I}^1, \mathcal{D}^1), \ldots, (\mathcal{I}^c, \mathcal{D}^c)$ and pair of error sets $\epsilon^1_1, \ldots, \epsilon^c_1$ and $\epsilon^1_2, \ldots, \epsilon^c_2$. C is a self-correcting program for f^1 with respect to the library with the same input set $(\mathcal{I}^1, \mathcal{D}^1), \ldots, (\mathcal{I}^c, \mathcal{D}^c)$ and with an error set $\epsilon^1, \ldots, \epsilon^c$, where for all $i = 1, \ldots, c$, $0 \le \epsilon^i_1 < \epsilon^i_2 \le \epsilon^i < 1$.*

As before, we require that both T and C be different than any correct program for f^1. To enforce this condition, we say that T and C are *different* than any correct program for f^1 if the incremental times of T and C, not including the time for calls to the programs $P^1, \ldots, P^c$, are smaller than the fastest known running time of any correct program for computing f^1. We say that T and C are *efficient* if the total time of T and C, including the time for the calls to the program $P^1, \ldots, P^c$, are within a constant multiplicative factor of the running time of P^1, assuming that the running times of $P^2, \ldots, P^c$ are reasonable with respect to the running time of P^1.

A typical way to build a self-testing/correcting pair (T, C) for f^1 with respect to a library f^1, f^2 is as follows. First, build a self-testing/correcting pair (T', C') for f^2. Now consider building the self-testing program T for f^1, where program P^1 supposedly computes f^1 and P^2 supposedly computes f^2. The typical situation is that T, in order to self-test P^1, needs to compute f^2 on various inputs. Instead of computing f^2 directly, T first uses T' to test how well P^2 computes f^2. If P^2 passes the test then T uses the self-corrector C' for f^2, which makes calls to P^2, to correctly compute f^2 whenever needed. Similarly, the self-corrector C may need to compute f^2 on various inputs, in which case it uses C' which in turn makes calls to P^2.

6 The Linear Algebra Library

We now show how to self-test/correct a library of possibly fallible programs for matrix multiplication, matrix inverse, determinant and rank. We use the following notation throughout this subsection.

Definition 6.1 (matrix notation) *Let $M_{n\times n}[F]$ be the set of $n \times n$ matrices with entries from a field F, and let $\mathcal{U}_{M_{n\times n}[F]}$ be the uniform distribution on $M_{n\times n}[F]$. For all $A \in M_{n\times n}[F]$, let* $\det(A)$ *be the determinant of A and let* $\operatorname{rank}(A)$ *be the rank of A. For all $r \in \{0,\ldots,n\}$, let $I^r_{n\times n}$ be the $n \times n$ matrix where all entries are 0 except that the first r entries along the main diagonal are 1, and thus $I^n_{n\times n}$ is the identity matrix. For all $r \in \{0,\ldots,n\}$, $M^r_{n\times n}[F]$ be the set of matrices in $M_{n\times n}[F]$ of rank r, and let $\mathcal{U}_{M^r_{n\times n}[F]}$ be the uniform distribution on $M^r_{n\times n}[F]$. Thus, $M^n_{n\times n}[F]$ is the set of invertible matrices in $M_{n\times n}[F]$.*

6.1 Matrix Multiplication

The input to matrix multiplication is $A, B \in M_{n\times n}[F]$, and the output is $A \cdot B$. The input to matrix inverse is $A \in M_{n\times n}[F]$, and the output is A^{-1} if it exists, and "NO" otherwise. The input to determinant is $A \in M_{n\times n}[F]$, and the output $\det(A)$. The input to rank is $A \in M_{n\times n}[F]$, and the output is $\operatorname{rank}(A)$.

For the analysis of the running time, we assume that field operations can be performed in constant time, and that an element from F can be randomly chosen uniformly in constant time. The self-testing/correcting pairs that we present are all *different* and *efficient*.

Program **Freivalds_Checker** described below is due to [20, Freivalds].

Specifications of Matrix_Mult Self-Correct(n, A, B, β):
If $\operatorname{error}(f, P, \mathcal{U}_{M_{n\times n}[F]} \times \mathcal{U}_{M_{n\times n}[F]}) \leq 1/8$ then the probability that the output is equal to $A \cdot B$ is at least $1 - \beta$.

Program Matrix_Mult Self-Correct(n, A, B, β)

```
Do for $i = 1, \ldots, \infty$
  Choose $A_1 \in_{\mathcal{U}} M_{n\times n}[F]$
  Choose $B_1 \in_{\mathcal{U}} M_{n\times n}[F]$
  $A_2 \leftarrow A - A_1$
  $B_2 \leftarrow B - B_1$
  $C \leftarrow P(A_1, B_1) + P(A_1, B_2) + P(A_2, B_1) + P(A_2, B_2)$
  If Freivalds_Checker$(n, A, B, C, \beta)$ ="PASS" then output $C$ and HALT
```

Specifications of Freivalds_Checker(n, A, B, C, β):
If $C \neq A \cdot B$ then output "FAIL" with probability at least $1 - \beta$. If $C = A \cdot B$ then output "PASS". The running time is $O(n^2 \ln(1/\beta))$.

Program Freivalds_Checker(n, A, B, C, β)

```
Do for $j = 1, \ldots, \ln(1/\beta)$
  Choose $n-$vector $R =< R_1, \ldots, R_n >$, where independently for each $i$, $R_i \in_{\mathcal{U}} F$
  If $C \cdot R \neq A \cdot (B \cdot R)$ then output "FAIL" and RETURN
Output "PASS"
```

Lemma 17 **Matrix_Mult Self-Correct** *meets the specifications. Furthermore, the expected total time is* $O(T(n) + n^2 \ln(1/\beta))$*, where* $T(n)$ *is the running time of* P *on inputs from* $M_{n \times n}[F] \times M_{n \times n}[F]$*.*

PROOF: $A_1 \in_{\mathcal{U}} M_{n\times n}[F]$, $A_2 \in_{\mathcal{U}} M_{n\times n}[F]$, $B_1 \in_{\mathcal{U}} M_{n\times n}[F]$, $B_2 \in_{\mathcal{U}} M_{n\times n}[F]$ and, although A_1 may depend on A_2 and B_1 may depend on B_2, A_1 and A_2 are independent of B_1 and B_2. Hence $P(A_i, B_j) \neq A_i \cdot B_j$ with probability at most 1/8, and thus $C = A \cdot B$ with probability at least 1/2 at each iteration. Let p be the probability that the final output of **Matrix_Mult Self-Correct** is equal to $A \cdot B$. With probability at least 1/2 in the first iteration $C = A \cdot B$, in which case **Freivalds_Checker** returns "PASS". With probability at most 1/2 in the first iteration, $C \neq A \cdot B$, in which case **Freivalds_Checker** returns "FAIL" with probability at least $1 - \beta$, and the second iteration starts. Thus $p \geq \frac{1}{2} + \frac{1}{2}(1-\beta)p$. From this, it can be verified that $1 - p$ is at most β.

The expected total time of **Matrix_Mult Self-Correct** is at most $O(T(n) + n^2 \ln(1/\beta))$ times the expected number of iterations until $C = A \cdot B$, which is at most two. ■

The self-testing program for matrix multiplication program is simple. The following step is executed $O(\ln(1/\beta))$ times to obtain a good estimate of error$(f, P, \mathcal{U}_{M_{n\times n}[F]} \times \mathcal{U}_{M_{n\times n}[F]})$. Independently choose $A \in_{\mathcal{U}} M_{n\times n}[F]$ and $B \in_{\mathcal{U}} M_{n\times n}[F]$ and set $C \leftarrow P(A, B)$. If the output of **Freivalds_Checker**$(n, A, B, C, 1/4)$ is "PASS", then the answer is 0 from the step, and if the output is "FAIL" then the answer is 1. It is easy to verify that if error$(f, P, \mathcal{U}_{M_{n\times n}[F]} \times \mathcal{U}_{M_{n\times n}[F]}) \geq 1/8$ then the fraction of 1 answers is at least 1/16 with probability at least $1 - \beta$, and if error$(f, P, \mathcal{U}_{M_{n\times n}[F]} \times \mathcal{U}_{M_{n\times n}[F]}) \leq 1/32$ then the fraction of 1 answers is at most 1/16 with probability at least $1 - \beta$. This yields a $(1/32, 1/8)$-self-tester for matrix multiplication.

6.2 Matrix Inversion

We next design a self-correcting program for matrix inversion. Hereafter, we call **Matrix_Mult Self-Correct** (abbreviated **MMSC**) whenever we want to multiply matrices together. The assumption is that **MMSC** uses a program P_1 has already been self-tested and "PASSED" to compute matrix multiplications. To avoid cluttering the explanation with messy details, we assume that P_1 "PASSED" for good reason, i.e. it has error probability at most 1/8, and thus **MMSC** does self-correct.

We use program **Gen_Inv_Matrix**(n) as a subroutine in our code to choose $A \in_{\mathcal{U}} M^n_{n\times n}[F]$. **Gen_Inv_Matrix**$(n)$ is due to [28, Randall], and a description of it can be found there. The incremental time of **Gen_Inv_Matrix**(n) is $O(n^2)$, excluding the time for computing the one required matrix multiplication. We assume that **Gen_Inv_Matrix**(n) calls **MMSC** in order to compute the matrix multiplication. Thus, **Gen_Inv_Matrix**(n) has a small probability of error, which we ignore for purposes of clarity. **Gen_Inv_Matrix/Det**(n), also due to [28, Randall], in addition to outputting $A \in_{\mathcal{U}} M^n_{n\times n}[F]$, also outputs $\det(A)$.

Specifications of Matrix_Inv Self-Correct(n, A, β):

If error$(f, P, \mathcal{U}_{M^n_{n\times n}[F]}) \leq 1/8$ and A is invertible then the output is A^{-1} with probability at least $1-\beta$. If A is not invertible then the output is "NO" with probability at least $1-\beta$.

Program Matrix_Inv Self-Correct(n, A, β)

```
    $N \leftarrow 12\ln(1/\beta)$
    Do for $i = 1, \ldots, N$
      $R \leftarrow$ Gen_Inv_Matrix$(n)$
      $R' \leftarrow$ MMSC$(n, A, R, 1/32)$
      $R'' \leftarrow P(R')$
      If $R''$ = "NO" then $answer_i \leftarrow$ "NO"
      Else
        $A' \leftarrow$ MMSC$(n, R, R'', 1/32)$
        If $I^n_{n\times n} \neq$ MMSC$(n, A, A', 1/32)$ then $answer_i \leftarrow$ "NO" else $answer_i \leftarrow A'$
    Output the most common answer among $\{answer_i : i = 1, \ldots, N\}$
```

Lemma 18 **Matrix_Inv Self-Correct** *meets the specifications.*

PROOF: Suppose that A is invertible. Then, because $R \in \mathcal{U}_{M^n_{n\times n}[F]}$, $A \cdot R \in_{\mathcal{U}} \mathcal{U}_{M^n_{n\times n}[F]}$. If the first call to **MMSC** is correct then $R' = A \cdot R$. Because the first call is correct with probability at least 31/32, the distance between the distribution on R' and $\mathcal{U}_{M^n_{n\times n}[F]}$ is at most 1/32. Consequently $R'' = P(R') = R'^{-1} = R^{-1} \cdot A^{-1}$ with probability at least $7/8 - 1/32$. If $R'' = R^{-1} \cdot A^{-1}$ and the second call to **MMSC** is correct then $A' = A^{-1}$. If the third call to **MMSC** is correct then $answer_i = A^{-1}$. Since these last two calls to **MMSC** are both correct with probability at least 15/16, $answer_i = A^{-1}$ with probability at least $7/8 - 1/32 - 1/16 \geq 3/4$. Now suppose that A is not invertible. Then, for every A', $I^n_{n\times n} \neq A' \cdot A$. Since the last call to **MMSC** is wrong with probability at most 1/32, it follows that $answer_i$ = "NO" with probability at least 31/32. Proposition 15 shows that $12\ln 1/\beta$ trials are sufficient to guarantee the result. ■

As was the case for the self-testing program for matrix multiplication, the self-tester for matrix inversion is simple. Notice that inputs need only be self-tested with respect to $\mathcal{U}_{M^n_{n\times n}[F]}$. The following step is executed $O(\ln(1/\beta))$ times to obtain a good estimate of error$(f, P, \mathcal{U}_{M^n_{n\times n}[F]})$. Set $R \leftarrow$ **Gen_Inv_Matrix**(n), and set $R' \leftarrow P(R)$. If $I^n_{n\times n} =$ **MMSC**$(R, R', 1/64)$ then the answer is 0 from the step, and otherwise the answer is 1. It is easy to verify that if error$(f, P, \mathcal{U}_{M^n_{n\times n}[F]})$ $\geq 1/8$ then the fraction of 1 answers is at least 1/16 with probability at least $1-\beta$, and if error$(f, P, \mathcal{U}_{M^n_{n\times n}[F]}) \leq 1/32$ then the fraction of 1 answers is at most 1/16 with probability at least $1-\beta$. This yields a $(1/32, 1/8)$-self-tester for matrix inversion.

6.3 Determinant

We next design a self-correcting program for determinant. Hereafter, we call **Matrix_Inv Self-Correct** (abbreviated **MISC**) whenever we want to find the inverse of a matrix. The assumption is that **MISC** uses a program P_2 has already been self-tested and "PASSED" to compute matrix inversions. To avoid cluttering the explanation with messy details, we assume that P_2 "PASSED" for good reason, i.e. it has error probability at most 1/8, and thus **MISC** does self-correct.

Specifications of Determinant Self-Correct(n, A, β):
If $\text{error}(f, P, \mathcal{U}_{M^n_{n\times n}[F]}) \leq 1/16$ then the output is $\det(A)$ with probability at least $1-\beta$.

Program Determinant Self-Correct(n, A, β)

```
$N \leftarrow O(\ln(1/\beta))$
Do for $i = 1, \ldots, N$
  If **MISC**$(n, A, 3/4)$ = "NO" then $answer_i \leftarrow 0$
  Else
    $R \leftarrow$**Gen_Inv_Matrix**$(n)$
    $R' \leftarrow$ **MMSC**$(n, A, R, 1/16)$
    $d_R \leftarrow P(R)$
    $d_{R'} \leftarrow P(R')$
    If $d_R = 0$ then $answer_i \leftarrow 0$ else $answer_i \leftarrow d_R/d_{R'}$
Output the most common answer among $\{answer_i : i = 1, \ldots, N\}$
```

One can easily prove the following lemma:

Lemma 19 Determinant Self-Correct *meets the specifications.*

As was the case for the self-testing program for matrix inversion, the self-tester for determinant is simple and the inputs need only be self-tested with respect to $\mathcal{U}_{M^n_{n\times n}[F]}$. The following step is executed $O(\ln(1/\beta))$ times to obtain a good estimate $\text{error}(f, P, \mathcal{U}_{M^n_{n\times n}[F]})$. Set $(R, d) \leftarrow$ **Gen_Inv_Matrix/Det**(n), and set $d' \leftarrow P(R)$. If $d = d'$ then the answer is 0 from the step, and otherwise the answer is 1. It is easy to verify that if $\text{error}(f, P, \mathcal{U}_{M^n_{n\times n}[F]}) \geq 1/8$ then the fraction of 1 answers is at least $1/16$ with probability at least $1-\beta$, and if $\text{error}(f, P, \mathcal{U}_{M^n_{n\times n}[F]}) \leq 1/32$ then the fraction of 1 answers is at most $1/16$ with probability at least $1-\beta$. This yields a $(1/32, 1/8)$-self-tester for matrix determinant.

6.4 Matrix Rank

We finally design a self-testing/correcting pair for Matrix Rank. One interesting aspect of the matrix rank self-corrector is that to self-correct an $n \times n$ matrix we call the program on $2n \times 2n$ matrices.

DEFINITION 6.2 (distribution for matrix rank) *Let $\mathcal{D}_n$ be the distribution defined by B randomly chosen as follows. Choose $r \in_{\mathcal{U}} \{0, \ldots, n\}$ and then choose $B \in_{\mathcal{U}} M^r_{n\times n}[F]$.*

Let $A \in M_{n\times n}[F]$.

Specifications of Matrix_Rank Self-Correct(n, A, β):
If $\text{error}(f, P, \mathcal{D}_{2n}) \leq 1/16$ then the output is $\text{rank}(A)$ with probability at least $1-\beta$.

Program Matrix_Rank Self-Correct(n, A, β)

```
$N \leftarrow O(\ln(1/\beta))$
Do for $i = 1, \ldots, N$
  Choose $r \in_{\mathcal{U}} \{0, \ldots, n\}$
  $A' \leftarrow \begin{bmatrix} A & I^0_{n\times n} \\ I^0_{n\times n} & I^r_{n\times n} \end{bmatrix}$
  $R \leftarrow$ **Gen_Inv_Matrix**$(2n)$
  $R' \leftarrow$ **MISC**$(2n, R, 1/16)$
  $S \leftarrow$ **MMSC**$(2n, A', R, 1/16)$
  $T \leftarrow$ **MMSC**$(2n, R', S, 1/16)$
  $answer_i \leftarrow P(T) - r$
Output the most common answer among $\{answer_i : i = 1, \ldots, N\}$
```

Lemma 20 Matrix_Rank Self-Correct *meets the specifications.*

PROOF: If the call to **MISC** and the two calls to **MMSC** are correct then $R' = R^{-1}$, $S = A' \cdot R$ and $T = R^{-1} \cdot A' \cdot R$ in which case $\text{rank}(T) = \text{rank}(A') = \text{rank}(A) + r$. Let $\mathcal{E}_{2n}$ be the distribution defined by B where B is randomly chosen as follows. Choose $r \in_{\mathcal{U}} \{0, \ldots, n\}$ and $B \in_{\mathcal{U}} M^{r+\text{rank}(A)}_{2n\times 2n}[F]$. Because $R \in_{\mathcal{U}} M^{2n}_{2n\times 2n}[F]$, we claim that the distribution $\mathcal{E}'_{2n}$ on T can be expressed in the form

$$\mathcal{E}'_{2n} = \frac{13}{16}\mathcal{E}_{2n} + \frac{3}{16}\mathcal{F}_{2n},$$

where $\mathcal{F}_{2n}$ is some distribution on $M_{2n\times 2n}[F]$. The case when T is chosen according to $\mathcal{E}_{2n}$ with probability $\frac{13}{16}$ corresponds to the case when each call to **MISC** and **MMSC** is correct, which happens with probability at least $\frac{13}{16}$ independent of R, and thus in addition $\text{rank}(T) = \text{rank}(A) + r$. It is not hard to verify that for all $B \in M_{2n\times 2n}[F]$, $\mathcal{E}_{2n}[\{B\}] \le 2\mathcal{D}_{2n}[\{B\}]$. From this and the assumption that $\text{error}(f, P, \mathcal{D}_{2n}) \le \frac{1}{16}$ it follows that

$$\Pr[P(B) \ne \text{rank}(B)] \le \frac{1}{8}$$

when B is randomly chosen according to $\mathcal{E}_{2n}$. From this and the fact that $\mathcal{E}'_{2n} = \frac{13}{16}\mathcal{E}_{2n} + \frac{3}{16}\mathcal{F}_{2n}$ it follows that

$$\Pr[P(T) \ne \text{rank}(A) + r] \le \frac{13}{16} \cdot \frac{1}{8} + \frac{3}{16} \le \frac{5}{16}.$$

Thus, for each i, $\Pr[answer_i = \text{rank}(A)] \ge \frac{11}{16} > \frac{1}{2}$. The lemma follows from a slight modification of Proposition 1. ■

As was the case for the self-testing program for matrix inversion, the self-tester for matrix rank is simple.

Specifications of Matrix_Rank Self-Test(n, β):

(1) If $\text{error}(f, P, \mathcal{D}_n) \le 1/64$ then the output is "PASS" with probability at least $1 - \beta$.

(2) If $\text{error}(f, P, \mathcal{D}_n) \ge 1/16$ then the output is "FAIL" with probability at least $1 - \beta$.

Program Matrix_Rank Self-Test(n, β)

```
$answer \leftarrow 0$
$N \leftarrow O(\ln(1/\beta))$
Do for $i = 1, \ldots, N$
  Choose $r \in_{\mathcal{U}} \{0, \ldots, n\}$
  $R \leftarrow$ **Gen_Inv_Matrix**$(n)$
  $R' \leftarrow$ **MISC**$(R, 1/256)$
  $S \leftarrow$ **MMSC**$(I^r_{n\times n}, R, 1/256)$
  $T \leftarrow$ **MMSC**$(R', S, 1/256)$
  $r' \leftarrow P(T)$
  If $r \neq r'$ then $answer \leftarrow answer + 1$
If $answer \geq N/32$ then output "FAIL" then output "PASS"
```

Lemma 21 Matrix_Rank Self-Test *meets the specifications.*

PROOF: It is easy to verify that if $\text{error}(f, P, \mathcal{D}_n) \geq 1/16$ then the fraction of 1 answers is at least $1/32$ with probability at least $1-\beta$ and if $\text{error}(f, P, \mathcal{D}_n) \leq 1/64$ then the fraction of 1 answers is at most $1/32$ with probability at least $1-\beta$. ■

7 Bootstrap Self-Testing

In this section we introduce another method of designing self-testers. It is easier to prove that this method of self-testing meets its specifications than it is for self-testing based on linearity. This method works for all the applications that the linear self-testing works for, as well as for polynomial multiplication, matrix multiplication, modular exponentiation when the ϕ function of the modulus is not known, and integer division. The drawback is that this method is often less efficient and that the code is slightly more complicated.

The two requirements for this method to work are random self-reducibility and:

DEFINITION 7.1 (smaller self-reducibility) *We say that f is* c-self-reducible to smaller inputs *if for all $x \in \mathcal{I}_n$, $f(x)$ can be expressed as an easily computable function F_{smaller} of $x, a_1, \ldots, a_c$ and $f(a_1), \ldots, f(a_c)$, where $a_1, \ldots, a_c$ are each in $\mathcal{I}_{n-1}$ and easily computable from x. Furthermore, for all $x \in \mathcal{I}_1$, $f(x)$ is easy to compute directly.*

For example, for integer multiplication, where $f(x_1, x_2) = x_1 \cdot x_2$, this condition is fulfilled as follows: Let $x = (x_1, x_2)$, where $x_1, x_2 \in \mathcal{Z}_{2^n}$ and where n is a power of two. Let x_1^L be the most significant half of the bits of x_1 and let x_1^R be the least significant half of the bits of x_1. Define x_2^L and x_2^R analogously with respect to x_2. Let $a_1 = (x_1^R, x_2^R)$, $a_2 = (x_1^L, x_2^R)$, $a_3 = (x_1^R, x_2^L)$ and $a_4 = (x_1^L, x_2^L)$. Then, $f(x) = F_{\text{smaller}}(x, a_1, \ldots, a_c, f(a_1), \ldots, f(a_c)) = f(a_1) + (f(a_2) + f(a_3))2^{n/2} + f(a_4)2^n$.

The overall idea behind this method is that once smaller size inputs have been self-tested, larger inputs can be self-tested by choosing a random input x, decomposing x into smaller inputs, self-correcting the smaller inputs using random self-reducibility (which works because smaller inputs have been self-tested), and then comparing the answer against the answer the program gives on input x. This method of bootstrapping can be continued until the desired input size is reached. We now give more specific details.

We say that $x \in \mathcal{I}_n$ is *bad* if $P(x) \neq f(x)$, and otherwise x is *good*. **Generic Self-Correct** is the program described on page 13. Program **Rec_Self-Test**, described below, verifies that most of the inputs in $\mathcal{I}_n$ are good given that, recursively, most of the inputs in $\mathcal{I}_{n-1}$ are good.

Specifications of Rec_Self-Test(n, β):

(1) If at least a fraction of $\frac{1}{4c}$ of the inputs in $\mathcal{I}_n$ are bad and at most a fraction of $\frac{1}{4c}$ of the inputs in $\mathcal{I}_{n-1}$ are bad then **Rec_Self-Test** outputs "FAIL" with probability at least $1-\beta$.

(2) If at most a fraction of $\frac{1}{16c}$ of the inputs in $\mathcal{I}_n$ are bad and at most a fraction of $\frac{1}{4c}$ of the inputs in $\mathcal{I}_{n-1}$ are bad then **Rec_Self-Test** outputs "PASS" with probability at least $1-\beta$.

Program Rec_Self-Test(n, β)

```
$N \leftarrow O(c \ln(1/\beta))$
Do for $m = 1, \ldots, N$
  $answer_m \leftarrow 0$
  Choose $x \in_{\mathcal{U}} \mathcal{I}_n$
  If $n = 1$ then:
    Compute $f(x)$ directly
    If $f(x) \neq P(x)$ then $answer_m \leftarrow 1$
  Else $n > 1$ then:
    Generate smaller inputs $a_1, \ldots, a_c$ based on $x$
    For $k = 1, \ldots, c$, $y_k \leftarrow$ Generic Self-Correct$(n-1, a_k, \frac{1}{16c^2})$
    If $F_{\text{smaller}}(x, a_1, \ldots, a_c, y_1, \ldots, y_c) \neq P(x)$ then $answer_m \leftarrow 1$
If $\Sigma_{k=1}^{N} answer_k / N \geq \frac{3}{16c}$ then "FAIL" else "PASS"
```

Lemma 22 Rec_Self-Test *meets the specification.*

PROOF:

(1) Because of the specifications for **Generic Self-Correct** and because it is called with confidence parameter $\frac{1}{16c^2}$, the probability that there is an incorrect y_k for $k = 1, \ldots, c$ is at most $\frac{1}{16c}$. Therefore, in each iteration $\Pr[answer_m = 1] \geq \frac{1}{4c}(1 - \frac{1}{16c}) \geq \frac{15}{64c} > \frac{3}{16c}$.

(2) In each iteration $\Pr[answer_m = 1] \leq \frac{1}{16c} + \frac{1}{16c} = \frac{2}{16c} < \frac{3}{16c}$.

Thus, the average of $answer_m$ over $O(c \ln(1/\beta))$ iterations is at least $\frac{3}{16c}$ with probability at least $1-\beta$ in case 1 and at most $\frac{3}{16c}$ with probability at least $1-\beta$ in case 2. ∎

Finally, we describe the main program **Generic Bootstrap Self-Test**. We make the convention that if any call to one of the subroutines returns "FAIL" then final output is "FAIL" and otherwise the output is "PASS".

Specifications of Generic Bootstrap Self-Test(l, x, β):

(1) If there is an i, $1 \leq i \leq l$, such that the fraction of of bad inputs in $\mathcal{I}_i$ is at least $\frac{1}{4c}$, then output "FAIL" with probability at least $1-\beta$.

(2) If for all i, $1 \le i \le l$, the fraction of bad inputs in $\mathcal{I}_i$ is at most $\frac{1}{16c}$ then output "PASS" with probability at least $1-\beta$.

Program Generic Bootstrap Self-Test(l, x, β)

For $i = 1, \ldots, l$, call **Rec_Self-Test**$(i, \beta/l)$.

Theorem 11 **Generic Bootstrap Self-Test** *meets the specifications.*

PROOF:

(1) If there is an i, $1 \le i \le l$ such that for all $1 \le j \le i-1$, the fraction of bad inputs in $\mathcal{I}_j$ is at most $\frac{1}{4c}$ and the fraction of bad inputs in $\mathcal{I}_i$ is at least $\frac{1}{4c}$ then **Rec_Self-Test**$(i, \beta/l)$ outputs "FAIL" with probability at least $1 - \beta/l \ge 1 - \beta$.

(2) If, for all i, $1 \le i \le l$, the fraction of bad inputs in $\mathcal{I}_i$ is at most $\frac{1}{16c}$ then **Rec_Self-Test**$(i, \beta/l)$ outputs "FAIL" with probability at most β/l. Thus, over the l calls, the probability that all answers are "PASS" is at least $1-\beta$.

■

7.1 Matrix Multiplication

We showed in Subsection 6.1 how to get a self-tester for matrix multiplication using **Freivalds_-Checker**. To illustrate the method, we show in this subsection how to get a self-tester based on bootstrapping. We retain the matrix notation introduced on page 28.

random self-reducibility: Let $A, B \in M_{n\times n}[F]$. Independently choose $A_1 \in_{\mathcal{U}} M_{n\times n}[F]$, $B_1 \in_{\mathcal{U}} M_{n\times n}[F]$ and let $A_2 \leftarrow A - A_1$, $B_2 \leftarrow B - B_1$. Then (A_1, B_1), (A_2, B_1), (A_1, B_2), (A_2, B_2) are each distributed according to $\mathcal{U}_{M_{n\times n}[F]} \times \mathcal{U}_{M_{n\times n}[F]}$ and $f(A,B) = f(A_1,B_1) + f(A_2,B_1) + f(A_1,B_2) + f(A_2,B_2)$.

smaller self-reducibility: Let $A, B \in M_{2n\times 2n}[F]$ where

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},\ B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

and $A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22} \in M_{n\times n}[F]$. Then

$$f(A,B) = \begin{pmatrix} f(A_{11},B_{11}) + f(A_{12},B_{21}) & f(A_{11},B_{12}) + f(A_{12},B_{22}) \\ f(A_{21},B_{11}) + f(A_{22},B_{21}) & f(A_{21},B_{12}) + f(A_{22},B_{22}) \end{pmatrix}.$$

Since matrix multiplication is randomly self-reducible and self-reducible to smaller inputs, the method of bootstrapping can be used to self-test the matrix multiplication function. The self-tester makes $O(\log(n))$ calls to the program. However, the self-tester makes only a constant number of the calls to the program on $n \times n$ matrices, only a constant number of the calls to the program are on $n/2 \times n/2$ matrices, etc. Thus, the incremental time of the self-tester is linear in the size of the input, and the total time is linear in the running time of the program.

7.2 Polynomial Multiplication

We consider multiplication of polynomials over finite fields: in this case $f(p, q) = p \cdot q$ where p, q are two degree n polynomials with coefficients from finite field F. Using Kaminski's polynomial multiplication result checker, one can get a self-tester for polynomial multiplication. We show how to get a self-tester based on the method of bootstrapping.

Let $\mathcal{P}_n[F]$ be the set of degree n polynomials where each coefficient is an element of the finite field F. Let U_n be the distribution on pairs of degree n polynomials where each coefficient is chosen independently and uniformly from the finite field F.

random self-reducibility: Let $p, q \in \mathcal{P}_n[F]$. Independently choose $p_1 \in_{\mathcal{U}} \mathcal{P}_n[F]$, $q_1 \in_{\mathcal{U}} \mathcal{P}_n[F]$, and let $p_2 \leftarrow p - p_1$, $q_2 \leftarrow q - q_1$. Then $(p_1, q_1), (p_2, q_1), (p_1, q_2), (p_2, q_2)$ are distributed according to U_n and $f(p, q) = f(p_1, q_1) + f(p_2, q_1) + f(p_1, q_2) + f(p_2, q_2)$.

smaller self-reducibility: Let $p, q \in \mathcal{P}_{2n}[F]$ where $p = p_1 x^n + p_2$, $q = q_1 x^n + q_2$, and $p_1, p_2, q_1, q_2 \in \mathcal{P}_n[F]$. Then $f(p, q) = f(p_1, q_1)x^{2n} + (f(p_1, q_2) + f(p_2, q_1))x^n + f(p_2, q_2)$.

Since polynomial multiplication is randomly self-reducible and self-reducible to smaller inputs, the method of bootstrapping can be used to self-test the polynomial multiplication function. The self-tester makes $O(\log n)$ calls to the program, and has incremental time linear in the size of the input, and the total time is linear in the running time of the program.

7.3 Modular Inverse

In this subsection, we develop some programs that are used in the modular exponentiation self-tester developed in the next subsection. For simplicity, we assume that we are using a correct program for modular multiplication in the code; all of the code can be modified to use the library approach described earlier, where all modular multiplications are computed by a self-correcting program that makes calls to a program for modular multiplication that has been self-tested.

Let R be a positive integer of length n. For $x \in \mathcal{Z}_R^*$, let $f(x, R)$ be the mod R inverse of x, i.e. $f(x, R) \cdot_R x = 1$. Let P be a program that supposedly computes f. We assume that P satisfies the following condition: When $x \in_{\mathcal{U}} \mathcal{Z}_R$, $P(x, R) \cdot_R x = 1$ with probability at least $\frac{1}{c \ln(n)}$ for some constant $c > 0$. We can easily estimate this probability by randomly choosing several independent $x \in_{\mathcal{U}} \mathcal{Z}_R$ and computing the fraction of these x that satisfy $P(x, R) \cdot_R x = 1$. For all $R > 3$, $\Phi(R) = |\mathcal{Z}_R^*| > \frac{R}{6 \ln(n)}$ [30, Rosser, Schoenfeld], and thus if P is correct for a constant fraction δ of the $x \in \mathcal{Z}_R^*$ then the above condition is true with $c = 6/\delta$.

We now describe a random generator **Gen_Inv_Mod**(R) which makes calls to P to generate $x \in_{\mathcal{U}} \mathcal{Z}_R^*$.

Function Gen_Inv_Mod(R)

```
Repeat forever
  Choose $x \in_{\mathcal{U}} \mathcal{Z}_R$
  Choose $y \in_{\mathcal{U}} \mathcal{Z}_R$
  $z \leftarrow x \cdot_R y$
  $z' \leftarrow P(z, R)$
```

```
    If $z' \cdot_R z = 1$ then return $x$ and EXIT
```

Lemma 23 *If* **Gen_Inv_Mod**(R) *returns* x, *then* $x \in_{\mathcal{U}} \mathcal{Z}_R^*$. *Furthermore, if* $P(w, R) \cdot_R w = 1$ *with probability at least* $\frac{1}{c\ln(n)}$ *when* $w \in_{\mathcal{U}} \mathcal{Z}_R$, *then the expected number of executions of the repeat loop before* **Gen_Inv_Mod**(R) *halts is* $O(c^2 \ln^2(n))$.

PROOF: $P(z, R) \cdot_R z = 1$ can be true only if $z \in \mathcal{Z}_R^*$, which in turn can only be true if both $x \in \mathcal{Z}_R^*$ and $y \in \mathcal{Z}_R^*$. The conditional probability of choosing x such that $x \in \mathcal{Z}_R^*$ is uniform. Furthermore, the conditional probability of choosing y such that $y \in \mathcal{Z}_R^*$ is uniform given x. Since the distribution defined by $x \cdot_R w$, where x is fixed in $\mathcal{Z}_R^*$ and $w \in_{\mathcal{U}} \mathcal{Z}_R^*$, is the uniform distribution $\mathcal{U}_{\mathcal{Z}_R^*}$, the conditional probability of choosing z such that $z \in \mathcal{Z}_R^*$ is uniform given $x \in \mathcal{Z}_R^*$. Thus, the probability that $P(z, R) \cdot_R z = 1$ is independent of x as long as $x \in \mathcal{Z}_R^*$. This implies that each $x \in \mathcal{Z}_R^*$ is equally likely to be the output of **Gen_Inv_Mod**(R).

The running time analysis is straightforward, noting that $x \in \mathcal{Z}_R^*$ with probability at least $\frac{1}{c\ln(n)}$, and independently $y \in \mathcal{Z}_R^*$ with probability at least $\frac{1}{c\ln(n)}$. ■

The incremental time of **Gen_Inv_Mod**(R), not counting the time for calls to the modular multiplication program, is $O(c^2 n \ln^2(n))$. The total time is $O(c^2 \ln^2(n) T(n))$, where $T(n)$ is the running time of the modular multiplication program.

We next develop a function that on input $x \in \mathcal{Z}_R^*$ and R outputs the mod R inverse of x. This function makes calls to both P and **Gen_Inv_Mod**. As before, we assume that P satisfies the condition described above.

Function Mod_Inv Self-Correct(x, R)

```
Repeat $O(c \ln(n))$ times
  $w \leftarrow$ Gen_Inv_Mod$(R)$
  $y \leftarrow x \cdot_R w$
  $y' \leftarrow P(y, R)$
  $z \leftarrow y' \cdot_R y$
  If $z = 1$ then EXIT repeat loop
If $z \neq 1$ then return $x' = 1$ else return $x' = w \cdot_R y'$
```

Mod_Inv Self-Correct (hereafter abbreviated **Mod_InvSC**) has the property that if $x \in \mathcal{Z}_R^*$ then with very high probability the output x' satisfies $x' \cdot_R x = 1$. For simplicity, hereafter we assume that if $x \in \mathcal{Z}_R^*$ then the $x' \cdot_R x = 1$ always.

The expected incremental time of **Mod_InvSc**(x, R) is $O(c^3 n \ln^3(n))$ and the total time is $O(c^3 \ln^3(n) T(n))$, where $T(n)$ is the running time of the modular multiplication program plus the running time of the modular inverse program.

7.4 Modular Exponentiation

Let R be a positive integer of length m and let $a \in \mathcal{Z}_R^*$. Let n be a positive integer that is a power of 2 and let $x \in \mathcal{Z}_{2^n}$. Let $f(a, x, R) = a^x \bmod R$. In previous sections we developed a self-testing/correcting pair for f when the factorization of R is known. In this subsection, we develop a self-testing/correcting pair for f without this assumption. Let P be a program that supposedly computes f. We make the convention that if the second argument in a call to P is 0 (i.e. the exponent is 0) then the call to P is not actually made and the answer is automatically set to 1.

Specifications of Mod_Expon Self-Correct(n, a, x, R, β):
If error$(f, P, \mathcal{U}_{\mathcal{Z}_R^*} \times \mathcal{U}_{\mathcal{Z}_{2^n}} \times \mathcal{U}_{\{R\}}) \leq 1/32$ then the output is $a^x \bmod R$ with probability at least $1 - \beta$.

Program Mod_Expon Self-Correct(n, a, x, R, β)

```
$N \leftarrow 12 \ln(1/\beta)$
For $i = 1, \ldots, N$ do
  Choose $x_1 \in_{\mathcal{U}} \mathcal{Z}_{2^n}$
  If $x_1 \leq x$ then $\delta \leftarrow 0$ else $\delta \leftarrow 1$
  $x_2 \leftarrow x - x_1 + \delta 2^n$
  Choose $x_3 \in_{\mathcal{U}} \mathcal{Z}_{2^n}$
  $x_4 \leftarrow 2^n - 1 - x_3$
  $b \leftarrow$ Gen_Inv_Mod$(R)$
  $\alpha_1 \leftarrow P(a \cdot_R b, x_1, R)$
  $\alpha_2 \leftarrow P(a \cdot_R b, x_2, R)$
  $\alpha_3 \leftarrow P(b, \delta x_3, R)$
  $\alpha_4 \leftarrow P(b, \delta x_4, R)$
  $\alpha_5 \leftarrow$ Mod_InvSC$(P(b, x_1, R), R)$
  $\alpha_6 \leftarrow$ Mod_InvSC$(P(b, x_2, R), R)$
  $\alpha_7 \leftarrow$ Mod_InvSC$(P(a \cdot_R b, \delta x_3, R), R)$
  $\alpha_8 \leftarrow$ Mod_InvSC$(P(a \cdot_R b, \delta x_4, R), R)$
  $answer_i \leftarrow \alpha_1 \cdot_R \alpha_2 \cdot_R \alpha_3 \cdot_R \alpha_4 \cdot_R \alpha_5 \cdot_R \alpha_6 \cdot_R \alpha_7 \cdot_R \alpha_8 \cdot_R (\delta a)$
Output the most common answer among $\{answer_m : m = 1, \ldots, N\}$
```

Lemma 24 **Mod_Expon Self-Correct** *meets the specifications.*

PROOF: It can be verified that $x_1 \in_{\mathcal{U}} \mathcal{Z}_{2^n}$, $x_2 \in_{\mathcal{U}} \mathcal{Z}_{2^n}$, $x_3 \in_{\mathcal{U}} \mathcal{Z}_{2^n}$ and $x_4 \in_{\mathcal{U}} \mathcal{Z}_{2^n}$. Furthermore, $b \in_{\mathcal{U}} \mathcal{Z}_R^*$, and from this and because $a \in \mathcal{Z}_R^*$, $a \cdot_R b \in_{\mathcal{U}} \mathcal{Z}_R^*$. Thus, in all eight calls to P the input distribution is $\mathcal{U}_{\mathcal{Z}_R^*} \times \mathcal{U}_{\mathcal{Z}_{2^n}} \times \mathcal{U}_{\{R\}}$ (except in the case when $\delta = 0$, in which case four of the calls to P are not actually made and the answer is automatically 1). Thus, with probability at least 3/4, all eight calls to P return the correct answer. It is not hard to verify that if all eight calls to P return the correct answer, then by the properties of **Mod_InvSC**, $answer_i = a^x \bmod R$. The lemma follows from Proposition 1. ■

Hereafter, we refer to **Mod_Expon Self-Correct** as **Mod_ExpSC**. The incremental time of **Mod_ExpSC**, not counting time for calls to the programs for modular multiplication and modular inverse, is $O(n + c^3 m \ln^3(m))$. The total time of **Mod_ExpSC** is $O(c^3 \ln^3(m) T(m) + T'(n, m))$, where $T(m)$ is the running time of the program for modular multiplication plus the running time of the program for computing modular inverse and $T'(n, m)$ is the running time of the program for computing modular exponentiation.

We now describe the recursive self-tester for modular exponentiation.

Specifications of Rec_Mod_Expon Self-Test(n, R, β):

(1) If error$(f, P, \mathcal{U}_{\mathcal{Z}_R^*} \times \mathcal{U}_{\mathcal{Z}_{2^{n/2}}} \times \mathcal{U}_{\{R\}}) \leq 1/32$ and error$(f, P, \mathcal{U}_{\mathcal{Z}_R^*} \times \mathcal{U}_{\mathcal{Z}_{2^n}} \times \mathcal{U}_{\{R\}}) \leq 1/128$ then the output is "PASS" with probability at least $1 - \beta$.

(2) If error$(f, P, \mathcal{U}_{\mathcal{Z}_R^*} \times \mathcal{U}_{\mathcal{Z}_{2^{n/2}}} \times \mathcal{U}_{\{R\}}) \leq 1/32$ and error$(f, P, \mathcal{U}_{\mathcal{Z}_R^*} \times \mathcal{U}_{\mathcal{Z}_{2^n}} \times \mathcal{U}_{\{R\}}) > 1/32$ then the output is "FAIL" with probability at least $1 - \beta$.

Program Rec_Mod_Expon Self-Test(n, R, β)

```
    $answer \leftarrow 0$
    $N \leftarrow O(\ln(1/\beta))$
    Do for $i = 1, \ldots, N$
      $b \leftarrow$ Gen_Inv_Mod$(R)$
      Choose $y \in_{\mathcal{U}} \mathcal{Z}_{2^n}$
      Let $y = y_1 2^{n/2} + y_2$, where $y_1, y_2 \in \mathcal{Z}_{2^{n/2}}$
      $\alpha_1 \leftarrow$ Mod_ExpSC$(n/2, b, y_1, 1/512)$
      $\alpha_2 \leftarrow$ Mod_ExpSC$(n/2, \alpha_1, 2^{n/2} - 1, 1/512) \cdot_R \alpha_1$
      $\alpha_3 \leftarrow$ Mod_ExpSC$(n/2, b, y_2, 1/512)$
      If $P(b, y, R) \neq \alpha_2 \cdot_R \alpha_3$ then $answer \leftarrow answer + 1$
    If $answer \geq N/64$ then output "FAIL" then output "PASS"
```

Lemma 25 Rec_Mod_Expon Self-Test *meets the specifications.*

PROOF: By design, $b \in_{\mathcal{U}} \mathcal{U}_{\mathcal{Z}_R^*}$ and $y \in_{\mathcal{U}} \mathcal{U}_{\mathcal{Z}_{2^n}}$. Because $b \in \mathcal{Z}_R^*$ and by the properties of **Mod_ExpSC**, $\alpha_1 \neq b^{y_1} \bmod R$ with probability at most $1/512$ independent of b and y_1. If $\alpha_1 = b^{y_1} \bmod R$, then $\alpha_1 \in \mathcal{Z}_R^*$. In this case, $\alpha_2 \neq \alpha_1^{2^{n/2}-1} \cdot_R \alpha_1 = b^{y_1 2^{n/2}} \bmod R$ with probability at most $1/512$. Similarly, $\alpha_3 \neq b^{y_2} \bmod R$ with probability at most $1/512$. Thus, the probability that $\alpha_2 \cdot_R \alpha_3 \neq b^y \bmod R$ is at most $3/512$. From this and Proposition 1 it can be verified that the lemma follows. ■

The incremental and total time of **Rec_Mod_Expon Self-Test** are linear in the incremental and total time of **Mod_ExpSC**(n, R, β), respectively.

We finally describe the self-tester for modular exponentiation, which is based on **Generic Boostrap Self-Test**. We make the convention that if any call to one of the subroutines returns "FAIL" then final output is "FAIL" and otherwise the output is "PASS".

Specifications of Mod_Expon Bootstrap Self-Test(n, R, β):

(1) If, for all $i = 1, \ldots, \log(n)$, error$(f, P, \mathcal{U}_{\mathcal{Z}_R^*} \times \mathcal{U}_{\mathcal{Z}_{2^i}} \times \mathcal{U}_{\{R\}}) \leq 1/128$ then output "PASS" with probability at least $1 - \beta$.

(2) If, for some $i = 1, \ldots, \log(n)$, error$(f, P, \mathcal{U}_{\mathcal{Z}_R^*} \times \mathcal{U}_{\mathcal{Z}_{2^i}} \times \mathcal{U}_{\{R\}}) > 1/32$ then output "FAIL" with probability at least $1 - \beta$.

Program Mod_Expon Bootstrap Self-Test(n, R, β)

```
    For $i = 1, \ldots, \log(n)$, call Rec_Mod_Expon Self-Test$(2^i, R, \beta/\log(n))$
```

Lemma 26 Mod_Expon Bootstrap Self-Test *meets the specifications.*

PROOF: Similar to the proof of Theorem 11 (page 35). ∎

The incremental and total time of **Mod_Expon Bootstrap Self-Test** are linear in the incremental and total time of **Mod_ExpSC**$(n, R, \beta/\log(n))$, respectively.

[1, Adleman Huang Kompella] have independently discovered a method of result checking the exponentiation function without the restriction that a and R be relatively prime. Their method uses similar ideas of testing by bootstrapping. The incremental time of their result checker is $O((n+m)\log(n))$, not counting calls to the modular multiplication program or the modular exponentiation program. The total time of their result checker is $O((T(n,m) + T'(n,m))\log(n))$, where $T(n,m)$ is the running time of the modular multiplication program for multiplying two n bit numbers mod a number of length m, and $T'(n,m)$ is the running time of the modular exponentiation program where both the base and modulus are of length m and the exponent is of length n.

8 Future Work

- Are there self-testing/correcting pairs for other important functions? In this paper, we have shown self-testing/correcting pairs for functions with the linearity property. Work in [6, Beaver Feigenbaum] and [26, Lipton] and [21, Gemmell Lipton Rubinfeld Sudan Wigderson] has extended this to functions which compute polynomials over finite fields and this has been extended in [34, Rubinfeld Sudan] to work over rational domains. A variety of other problems, have result checkers, and thus also self-testers. It would be interesting to find self-correctors for such problems. An example is sorting.
- Is it possible to show that some functions are not going to have a self-testing/correcting pair? Some progress can be found in [18, Feigenbaum Kannan Nisan], [39, Yao], [7, Beigel Feigenbaum].
- Are there applications of the combinatorial theorems introduced in this paper in other areas? We suggest the development of more probabilistic tools long these lines. Significant progress in this direction has been made in a series of papers by [21, Gemmell Lipton Rubinfeld Sudan Wigderson].
- One area of practical concern for self-testing/correcting pairs is the overhead incurred by running the self-tester and self-corrector. Recently a *batch self-corrector* for any function with the linearity property has been designed which reduces the overhead to a small additive factor if it is infrequent that P answers incorrectly for some input in a batch [32, Rubinfeld]. We would like to design batch self-correctors for other important functions.

9 Acknowledgements

We thank Silvio Micali for pointing out the general applicability of our methods, for his enthusiastic support and for numerous illuminating technical discussions. We thank Don Coppersmith for allowing us to use his elegant version of the proofs of Theorems 3 and 4. We thank Oded Goldreich, Shafi Goldwasser, Sampath Kannan, Richard Cleve, Don Coppersmith, Michael Ben-Or, Russell Impagliazzo and Steve Omohundro for very helpful discussions.

References

[1] Adleman, L., Huang, M., Kompella, K., "Efficient Checkers for Number-Theoretic Computations", Submitted to *Information and Computation.*

[2] Aho, A., Hopcroft, J., Ullman, J., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachussetts, 1974.

[3] Babai, L., "Trading Group Theory for Randomness", proceedings of the 17^{th} *ACM Symposium on Theory of Computing*, pp. 421–429, 1985.

[4] Babai, L., "E-mail and the power of interaction", proceedings of the 5^{th} *Structures in Complexity Theory Conference*, 1990.

[5] Babai, L., Fortnow, L., Lund, K., "Non-Deterministic Exponential Time has Two-Prover Interactive Protocols", proceedings of the 31^{st} *IEEE Symposium on Foundations of Computer Science*, 1990.

[6] Beaver, D., Feigenbaum, J., "Hiding Instance in Multioracle Queries", proceedings of *Symposium of Theoretical Aspects of Computer Science*, 1990.

[7] Beigel, R., Feigenbaum, J., "On the Complexity of Coherent Sets", *AT&T Technical Memorandum*, February 19, 1990.

[8] Ben-Or, M., Coppersmith, D., Luby, M., Rubinfeld, R., "Convolutions on Groups", in preparation.

[9] Ben-Or, M., Goldwasser, S., Kilian, J., and Wigderson, A., "Multi-Prover Interactive Proofs: How to Remove Intractability", proceedings of the 20^{th} *ACM Symposium on Theory of Computing*, pp. 113-131, 1988.

[10] Blum, M., "Designing programs to check their work", *ICSI technical report TR-88-009.*

[11] Blum, M., Raghavan, P, "Program correctness: can one test for it?", *Information Processing 89*, G.X. Ritter (ed.), Elsevier Science Publishers B.V. (North-Holland), IFIP 1989, pp. 127-134.

[12] Blum, M., Kannan, S., "Designing programs that check their work," proceedings of the 21^{st} *ACM Symposium on Theory of Computing*, 1989.

[13] Blum, M., Luby, M., Rubinfeld, R., "Program Result Checking Against Adaptive Programs and in Cryptographic Settings", presented in *Distributed Computing and Cryptography workshop*, 1989, paper appears in *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, vol. 2, 1991.

[14] Blum, M., Luby, M., Rubinfeld, R., "Self-Testing/Correcting with Applications to Numerical Problems," proceedings of 22^{nd} *ACM Symposium on Theory of Computing*, 1990.

[15] Blum, M., and Micali, S., "How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits", preliminary version appears in proceedings of 23^{rd} *IEEE Symposium on Foundations of Computer Science*, 1982, journal paper appears in *SIAM J. on Computing*, Vol. 13, 1984, pp. 850-864,

[16] Cleve, R., Luby, M., "A Note on Self-Testing/Correcting Methods for Trigonometric Functions", *ICSI technical report TR-90-032*, 1990.

[17] Coppersmith, D., Winograd, S., "Matrix Multiplication via Arithmetic Progressions", proceedings of the 19^{th} *ACM Symposium on Theory of Computing*, 1987.

[18] Feigenbaum, J., Kannan, S., Nisan, N., "Lower Bounds on Random Self-Reducibility", proceedings of the 5^{th} *Structures in Complexity Theory Conference*, 1990.

[19] Fortnow, L., Karloff, H., Lund, K., Nisan, N., "The Polynomial Hierarchy has Interactive Proofs", proceedings of the 31^{st} *IEEE Symposium on Foundations of Computer Science*, 1990.

[20] Freivalds, R., "Fast Probabilistic Algorithms", Springer Verlag Lecture Notes in CS No. 74, Mathematical Foundations of CS, 57-69 (1979).

[21] Gemmell, P., Lipton, R., Rubinfeld, R., Sudan, M., Wigderson, A., "Self-Testing/Correcting for Polynomials and for Approximate Functions", proceedings of the 23^{rd} *ACM Symposium on Theory of Computing*, 1991.

[22] Goldwasser, S., Micali, S. and Rackoff, C., "The Knowledge Complexity of Interactive Proof Systems," a preliminary version appeared in 17^{th} *ACM Symposium on Theory of Computing*, 1985. final version appears in *SIAM J. on Computing*, Vol. 18, No. 1, 1989, pp. 186-208.

[23] Kaminski, Michael, "A note on probabilistically verifying integer and polynomial products," *JACM*, Vol. 36, No. 1, January 1989, pp.142-149.

[24] Kannan, S., "Program Result Checking with Applications", Ph.D. thesis, U.C. Berkeley, 1990.

[25] Karp, R., Luby, M., Madras, N., "Monte-Carlo Approximation Algorithms for Enumeration Problems," *J. of Algorithms*, Vol. 10, No. 3, Sept. 1989, pp. 429-448.

[26] Lipton, R., "New directions in testing", presented in *Distributed Computing and Cryptography workshop*, 1989, paper appears in *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, vol. 2, 1991, pp. 191–202.

[27] Nisan, N., "Co-SAT Has Multi-Prover Interactive Proofs", e-mail announcement, November 1989.

[28] Randall, D., "Efficient Random Generation of Nonsingular Matrices", *U.C. Berkeley Technical Report*, Computer Science Department, No. 91/658, 1991.

[29] Rènyi, A., (1970), **Probability Theory**, North-Holland, Amsterdam.

[30] Rosser, J.B., Schoenfeld, L., "Approximate formulas for some functions of prime numbers", Illinois J. Math, 6, 1962, pp.64-94.

[31] Rubinfeld, R., "Designing Checkers for Programs that Run in Parallel", *ICSI technical report TR-90-040.*

[32] Rubinfeld, R. "Batch Checking Linear Functions", manuscript, 1990.

[33] Rubinfeld, R. "A Mathematical Theory of Self-Checking, Self-Testing and Self-Correcting Programs", Ph.D. thesis, Computer Science Department, U.C. Berkeley,, 1990.

[34] Rubinfeld, R., Sudan, M., "Self-Testing Polynomial Functions Efficiently and over Rational Domains", proceedings of *ACM SIAM Symposium on Discrete Algorithms*, 1992.

[35] Schönhage, A., personal communication through Michael Fischer.

[36] Schönhage, A., Strassen, V., "Schnelle Multiplikation grosser Zahlen," *Computing* 7, 281-292.

[37] Shamir, A., "IP = PSPACE", proceedings of the 31st *IEEE Symposium on Foundations of Computer Science*, 1990.

[38] Strassen, V., "Gaussian Elimination is not Optimal", Numerische Mathematik, 13, 1969, pp. 354-356.

[39] Yao, A., "Coherent Functions and Program Checking", proceedings of the 22nd *ACM Symposium on Theory of Computing*, 1990.