# Testing Multivariate Linear Functions: Overcoming the Generator Bottleneck

Funda Ergün *

November 11, 1994

**Abstract**

The problem of testing program correctness has received considerable attention in computer science. One approach to this problem is the notion of self-testing programs [BLR90]. Self-testing usually becomes more costly in the case of testing multivariate functions. In this paper we present efficient methods for self-testing multivariate linear functions. We then apply these methods to several multivariate linear problems to construct efficient self-testers.

*Cornell University. email: ergun@cs.cornell.edu. This work is supported by ONR Young Investigator Award N00014-93-1-0590

# 1 Introduction

Self-testing/correcting programs, which were introduced in [BLR90], are a powerful tool for attacking the problem of program correctness. Various problems have been shown to have self-testers and self-correctors[BLR90][BF90][Lip91][CL90][GLRSW91][RS92][RS93]. In this paper we investigate the problem of self-testing *multivariate* linear functions, i.e., given a multivariate linear function $f$ and a program $p$ that claims to compute $f$, we try to determine efficiently whether $p$ computes $f(x)$ for most inputs $x$. [BLR90] investigates the characterization of functions by their properties. In particular, they characterize some *univariate* linear functions and use this characterization for self-testing these functions in the following manner: Let $f$ be the univariate linear function to be computed, and $p$ be the program to be self-tested, where $x$ is selected from a group. $p$ is first tested for linearity: If $p(x+y) = p(x) + p(y)$ for most choices of random $x$, $y$, then there must exist a function $g$ which is usually equal to $p$ on most inputs and which is always linear. Once such a linear function $g$ is determined to exist one must determine that it is the *right* linear function. To achieve this, they perform the *neighbor test*, which ensures that $g(1) = f(1)$, therefore $g$ h! as the correct slope. Since 1 is

One way to get around this generator bottleneck has been to exploit the property of "downward self-reducibility" [BLR90] . However, self-testers that use this property have to make $\log n$ or more calls to the program, depending on the way that the problem decomposes into similar problems of smaller size. For instance, a tester for the "permanent" function has to make $O(n)$ calls to the program being tested.

Our contribution in this paper is to introduce simple and efficient techniques to get around the generator bottleneck in testing linear multivariate problems. We use a method similar to that used to test the univariate functions. We exploit the linearity of the functions and find a small and constant number of other properties that uniquely define these functions. However, we do not test the function's correctness at all of the generators (in the examples that we present, there are $O(n)$ or $O(n^2)$ of them); rather, we test the function at one generator, and then, explore the ways of "generating" the other generators from the generator already tested. For instance, for polynomials, the generators are 1, $x$, $x^2, x^3$, etc. It can be seen that all these generators can be generated from the first one, 1, by multiplying it by $x$ a number of times. In general, since we keep our variables in vector and matrix form, this generation usually involves "shifting" or "rotating" the input in some direction. For instance, if we keep the coefficients of a polynomial in a vector, we can get the above generators from the first one by shifting the contents of the vector to the left (we should be careful abo! ut not losing any values at the l

We apply the techniques that we develop to two multivariate linear functions that do not yet have self-testers: the two directions of the conversion between polynomial representations in Fast Fourier Transformation, i.e., coefficient-to-point value representation and point-value to coefficient representation. Another problem that we apply them to is polynomial multiplication. [BLR90] gives a bootstrap tester for this problem that makes $\log n$ calls to the program being tested. The advantages of using our method to test this problem are that the number of tests needed is fixed and small, the tests themselves are simple, and each test makes only a constant number of calls to the program itself. Finally, we give a new self-tester for matrix multiplication. It is already known that matrix multiplication can be tested with a constant number of calls to the program using a result checker due to Freivalds [Fre79], and a boostrap self-tester for this p! roblem is given in

In all of our examples there are at least $O(n)$ generators, where $n$ is the degree of the polynomial where the input is a polynomial, or the size of the vector or the square matrix when the inputs are vectors or matrices respectively. The functions are defined over multiple variables from a finite field $\mathcal{Z}_p$, but they can be extended to rational domains using standard techniques [GLRSW91] [RS93]. All of our self-testers make a constant number of calls to the programs regardless of the problem size. Since the tests are simple, the running time of each test is dominated by the calls to the program, and since there is a constant number of tests, the running times of these self-testers are the same as those of the programs that they are testing.

## 2 Organization of the Paper

The next section contains the the definitions of some important relevant concepts and some global variables that we refer to in the self-testers presented. The rest of the paper shows how our method can be applied to construct self-testers for various problems: Section 4 shows how to construct a self-tester for Fast Fourier Transform, Section 5 for polynomial multiplication and Section 6 for matrix multiplication.

## 3 Definitions

A *self-tester* is a probabilistic oracle program that determines with high probability whether a program P computes a given function $f$ correctly on most inputs. $T_f$ should be different from $p$, and as efficient as possible[BK89]. More formally, an $(\epsilon_1, \epsilon_2)$-self-testing program $T_f$ returns for any program $p$:

1. PASS with probability $\geq 1 - \delta$ if $Pr_x[p(x) = f(x)] \leq \epsilon_1$
2. FAIL with probability $\geq 1 - \delta$ if $Pr_x[p(x) = f(x)] \geq \epsilon_2$

A self-correcting program, takes a program $p$ that computes a function $f$ correctly on most of its inputs and turns it into a program that computes $f$ correctly with arbitrarily high probability on every input. and A self-corrector can be used in conjunction with a self-tester to first test whether a program is correct on most of the inputs and then self-correct it so that it becomes correct with the desired probability.

We say that $f$ has the linearity property if $f(x_1 \circ x_2) = f(x_1) \circ' f(x_2)$, where $x_1, x_2$ come from a field $\circ$ and $\circ'$ are two field operations.

For all the programs that we give, we define these global variables:

$m = \ln(2/\delta)$

$q = \ln(6/\delta)(1 - \epsilon/2) + 1/\sqrt{2}$

$l = \lceil \log_2(1/\epsilon) \rceil$

$l_2 = \lceil \log_2(2/\delta) \rceil$

## 4 Fast Fourier Transformation

The Fast Fourier Transformation (FFT) involves performing conversions between the coefficient and point-value representations of polynomials. Here we present testers for conversions in both directions.

## 4.1 From Coefficient to Point-Value Representation

**Definitions:**

Let $f : (\mathcal{Z}_p)^{n+1} \to (\mathcal{Z}_p)^{n+1}$ be the function that converts the coefficient representation of a polynomial $Q(z)$ to its point-value representation. In this context, in the equation $\overline{y} = f(\overline{a})$, $\overline{a}$ is the size $n+1$ vector containing the $n+1$ coefficients of the input polynomial, and $\overline{y}$ is the size $n+1$ vector containing the $y$-coordinates of the $n+1$ points in the point-value representation of the same polynomial, where the $n+1$ $x$-values are fixed. $y_i$ refers to the $i^{th}$ entry in vector $\overline{y}$.

For the sake of simplicity, we assume WLOG throughout this test that when the program $p$ is given an input of size less than $n+1$, it will pad the vector with 0's from the left until the size of the vector is exactly $n+1$, and then compute the function.

We define *shift-left* operation as shifting the contents of the coefficient vector to the left by 1, and making the rightmost element 0. In the context of this paper, we will perform this operation on size $n$ vectors, obtaining size $n+1$ vectors as the result. This is equivalent to multiplying the polynomial represented in this vector by $z$. Therefore, the coefficient vector of $Q(z) \cdot z$ is denoted by shl( $\overline{a}$),where $\overline{a}$ is the representation of the polynomial $Q(z)$. $\overline{y} \odot \overline{x}$ refers to the vector $[x_0 y_0, x_1 y_1, \ldots, x_n y_n]$, where the $x_i$'s are the $n+1$ fixed $x$-coordinates. Note that if $\overline{b}$ and $\overline{c}$ are the coefficient and point-value representations of the same vector respectively, then so are shl($\overline{c}$) and $\overline{c} \odot \overline{x}$.

We define $\overline{r} \in (\mathcal{Z}_p)^{n+1}$ to be the vector $[0, 0, 0, \ldots, 0, 1]$.

We also define:

$$g(\overline{a}) \equiv \mathrm{maj}_{\overline{b} \in (\mathcal{Z}_p)^{n+1}} \{p(\overline{a} - \overline{b}) + p(\overline{b})\}.$$

**The Test:**

The first test checks whether $p$ is linear most of the time. The second one attempts at guessing $g(\overline{r})$ and checks whether it is the same as $f(\overline{r})$. The $n$ generators in this problem are 1, $x$, $x^2$, etc. $\overline{r}$, which is the vector $[0, 0, \ldots, 0, 1]$, represents the first one of these generators and the the others are represented by the size $n+1$ vectors $[0, \ldots, 1, 0]$, $[0, \ldots, 1, 0, 0]$, etc. All the generators can be obtained by shifting $\overline{r}$ to the left (which in effect is the same as multiplying with $z$) a number of times. The point-value representation of a polynomial $z \cdot Q(z)$ can be computed easily if that of $Q(z)$ is known. Test 3 ensures that $g$ is *consistent with respect to shift-left operation*, i.e., given a vector of coefficients $\overline{a}$ and a vector $g(\overline{a})$ that contains the $y$-values of the point-value representation of $\overline{a}$, $\overline{a}$ is shifted left once, and the test checks that the new result that $g$ returns for this new vector is consistent with $g(\overline{a})$. More precisely, since the polynomial is multiplied by $z$, all the $y$-values in $g(\overline{a})$ should be multiplied with their respective $x$-values in the new result, yielding $g(\overline{a}) \odot \overline{x}$. If we determine that $g$ is always consistent with respect to the shift left operation, then, combined with the information that it is correct at the first generator , we can conclude by induction on the number of left shifts that it must be correct at all the generators.

```
/*Test 1*/
count = 0;
for i = 1...m do
    pick random ā, b̄ ∈ (Z_p)^{n+1}
    if p(ā) + p(b̄) = p(ā + b̄) then count = count +1;
if count < q then FAIL.

/*Test 2*/
for i = 1...l do
   pick random ā ∈ (Z_p)^{n+1}
    if f(r̄) ≠ p(ā) + p(r̄ − ā) then FAIL

/*Test 3*/
count = 0;
for i = 1...m do
   pick random ā ∈ (Z_p)^n
      for j = 1...l + 2 do
         pick random b̄ ∈ (Z_p)^n
         g_j(ā) = p(b̄) + p(ā − b̄)
       if all of the g_j's are the same then ȳ = g_1 else FAIL
      for j = 1...l + 2 do
         pick random b̄ ∈ (Z_p)^{n+1}
         g_j(shl(ā)) = p(b̄) + p(shl(ā) − shl(b̄))
       if all of the g_j's are the same then ȳ_2 = g_1 else FAIL
      if ȳ_2 = ȳ ⊙ x̄ then count = count + 1
if count ≥ q then PASS else FAIL
```

The following lemma states that the properties that this test checks for are necessary and sufficient to uniquely define $f$.

**Lemma 1** *If $g$ is always linear, computes $\overline{r}$ correctly, and is consistent with respect to shifting left of its argument, then $g$ computes the same function as $f$.*

**Proof:** $g$ is the same as $f$ on one input, namely the first generator $\overline{r}$. Because of consistency with respect to shift-lefts and property 3, $g$ is the same as $f$ on all generators. Because $g$ is always linear, we can generate all other members of the field using these generators. Therefore $g$ has to be the same as $f$. $\square$

If the program can pass tests 1 and 3, then it has two of the necessary properties:

**Lemma 2** *If $\epsilon < 1/2$ and $p$ is linear with probability at least $1-\epsilon/2$, then $g$ is always linear. If $g$ is consistent with respect to shifting left of its input on $1-\epsilon$ of its inputs, then it is always consistent with respect to shift-lefts.*

**Proof:** The proof for the first property is given in [BLR90], therefore we only give a proof for the second one. Assume $\overline{a} \in (\mathcal{Z}_p)^n$:

$$
\begin{aligned}
\Pr_{\overline{b}\in(\mathcal{Z}_p)^{n+1}}[g(\overline{a}) &= g(\overline{b}) + g(\overline{a}-\overline{b}) \\
&= g(\overline{d}) + g(\overline{c}) \\
&= g(shr(\overline{d}))\odot\overline{x} + shr(g(\overline{c}))\odot\overline{x} \\
&= g(shr(\overline{d}) + shr(\overline{c}))\odot\overline{x} \\
&= g(shr(\overline{a}))\odot\overline{x}] \qquad\qquad > 1-2\epsilon > 0
\end{aligned}
$$

In other words, $g(shl(\overline{a})) = g(\overline{a})\odot\overline{x}$. The operation shr is *shift-right*, which denotes a shifting right of the contents of a vector, deleting the now empty leftmost place. $\overline{d}$ is the vector $\overline{b}$ with the $0^{th}$ entry zeroed, and $\overline{c}$ is the vector $\overline{a}-\overline{b}$ with the $0^{th}$ entry zeroed. Note that they add up to $\overline{a}$ and that $shr(\overline{d})$ and $shr(\overline{c})$ are random size $n$ vectors.

The first, second, and fourth equalities follow from the linearity of $g$, and hold with probability 1. The third one follows from the consistency of $g$ with respect to shifting left with high probability, and holds with probability $1-2\epsilon$. The last one is just a rewriting of the vectors involved. Since the first and the last terms are independent of $\overline{b}$, and the equality holds with probability strictly greater than 0, then it has to hold with probability 1. □

Since Test 2 checks directly whether $g$ is correct at $\overline{r}$, if the program passes all three of the tests, with high probability it has the three properties mentioned in Lemma 1.

**Theorem 3** *The above program is a $(0,\epsilon)$-self-tester for the coefficient-to-point-value conversion of FFT.*

**Proof:**

1. It is easy to see that if $p$ is the same as $f$ on all inputs, the program is going to pass all the tests with probability 1.
2. If $p$ is wrong with probability $> \epsilon$:

   Using the argument from [BLR90], we know that $p$ is equal to $g$ on at least $1-\epsilon$ of the inputs. Therefore, if $p$ is not $\epsilon$-correct, $g$ cannot be equal to $f$ for all inputs, which means it has to violate at least one of the properties mentioned in Lemma 1. It can be shown using Chernoff bounds that if $p$ is not correct at least $1-\epsilon$ of the time, it will fail the test with probability at least $1-\delta$.

□

### 4.2 From Point-Value to Coefficient Representation

This conversion can be tested in a very similar way to the testing of the coefficient-to-point-value conversion. We present the third test only; the first two are similar to Test 1 and Test 2 of the program for testing coefficient to point value representation conversion, although test 2 checks for a different base case. It can be shown with methods similar to those in section 4.1 that the program consisting of the two tests mentioned and the one below is a $(1,\epsilon)$-self-tester for point-value to coefficient conversion.

```
/*Test 3*/
count = 0;
for i = 1...m do
    pick random ā ∈ (Z_p)^{n+1}
        for j = 1...l + 2 do
            pick random b̄ ∈ (Z_p)^n
            g_j(ā) = p(b̄) + p(ā − b̄)
        if all of the g_j's are the same then ȳ = g_1 else FAIL
        for j = 1...l + 2 do
            pick random b̄ ∈ (Z_p)^{n+1}
            g_j(ā) = p(b̄) + p(ā ⊙ x̄ − b̄)
        if all of the g_j's are the same then ȳ_2 = g_1 else FAIL
        if ȳ_2 = shl(ȳ) then count = count + 1
if count ≥ q then PASS else FAIL
```

## 5 Polynomial Multiplication

**Definitions:** Let the vector $\overline{a} = [a_0, a_1, \ldots, a_n]$ denote the degree $n$ polynomial $a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$. Let $p(\overline{a})$ be the function computed by the program that claims to be doing the polynomial multiplication. Define:

$$g(\overline{a}, \overline{b}) \equiv \mathrm{maj}_{\overline{c},\overline{d} \in (\mathcal{Z}_p)^n} \{P(\overline{a} - \overline{c}, \overline{d}) + P(\overline{c}, \overline{d}) + P(\overline{a} - \overline{c}, \overline{b} - \overline{d}) + P(\overline{c}, \overline{b} - \overline{d})\}$$

**The Test:**

The generators are the same as in the FFT self-tester, namely $[0, 0, \ldots, 0, 1], [0, 0, \ldots, 1, 0], etc.$ Therefore we use a self-tester similar to the ones for FFT. The first test checks for linearity, and the second one tries to ensure that $g(\overline{a}, \overline{r}) = f(\overline{a}, \overline{r})$ for most $\overline{a}$, where $\overline{r}$ is the first generator. The third one tries to ensure that most of the time when the second argument of $g$ is shifted left (multiplied with $z$), the result is shifted left as well.

```
/*Test 1*/
count = 0;
for i = 1...m do
    pick random ā, b̄, c̄, d̄ ∈ (Z_p)^{n+1}:
    if p(ā, c̄) + p(ā, d̄) + p(b̄, c̄) + p(b̄, d̄) = p(ā + b̄, c̄ + d̄) then count = count +1;
if count < q then FAIL.
/*Test 2*/
count = 0;
```

*Continued on next page*

*Continued from page 6*

```
for i = 1...m do
    pick random ā ∈ (Z_p)^{n+1}
    for j = 1...l_2 do
        pick random b̄, c̄ ∈ (Z_p)^{n+1}
        g_j(r̄, ā) = p(b̄, c̄) + p(r̄ − b̄, c̄) + p(b̄, ā − c̄) + p(r̄ − b̄, ā − c̄)
    if all of the g_j'a are the same then ȳ = g_1 else FAIL
     if ȳ = f(r̄, ā) then count = count + 1
if count < q then FAIL

/*Test 3*/
count = 0;
for i = 1...m do
    pick random ā ∈ (Z_p)^{n+1} and b̄' ∈ (Z_p)^n
    b̄= b̄' concatenated with a 0 from the left
        for j = 1...l + 2 do
            pick random c̄, d̄ ∈ (Z_p)^{n+1}
            g_j(ā, b̄) = p(c̄, d̄) + p(ā − c̄, d̄) + p(ā, b̄ − d̄) + p(ā − c̄, b̄ − d̄)
         if all of the g_j's are the same then ȳ_1 = g_1 else FAIL
        for j = 1...l + 2 do
            pick random c̄, d̄ ∈ (Z_p)^{n+1}
            g_j(ā, shl(b̄)) = p(c̄, d̄) + p(ā − c̄, d̄) + p(ā, shl(b̄) − d̄) + p(ā − c̄, shl(b̄) − d̄)
         if all of the g_k's are the same then ȳ_2 = g_1 else FAIL
        if ȳ_2 = shl(ȳ_1) then count = count + 1
if count ≥ q then PASS else FAIL
```

The following lemma explains that Test 2 actually checks whether $g$ is correct every time its first input is $\overline{r}$:

**Lemma 4** *If $\epsilon < 1/2$ and $g(\overline{a}, \overline{r}) = f(\overline{r}, \overline{a})$ for at least $\geq 1 - \epsilon$ of the $\overline{a}$'s, then $g(\overline{r}, \overline{a}) = f(\overline{r}, \overline{a})$ for all $\overline{a}$'s.*

**Proof:**

$$
\begin{aligned}
\Pr_{\overline{b}}[g(\overline{r}, \overline{a}) &= g(\overline{r}, \overline{b}) + g(\overline{r}, \overline{a} - \overline{b}) \\
&= f(\overline{r}, \overline{b}) + f(\overline{r}(\overline{a} - \overline{b}) \\
&= f(\overline{r}, \overline{a})] \qquad > 1 - 2\epsilon > 0
\end{aligned}
$$

The first equality holds from the linearity of $g$, with probability 1. The second equality holds with probability $> 1 - 2\epsilon$, due to the assumption, and the third one holds all the time from the linearity of polynomial multiplication. Since the first and last terms are equal with nonzero probability and they do not contain $\overline{b}$, they must be equal with probability 1. □

Test 3 also determines a general property of $g$:

**Lemma 5** *If $\epsilon < 1/4$ and $g$ is consistent with respect to shifting left with probability $\geq (1 - \epsilon)$, then $g$ is always consistent with respect to shifting left.*

**Proof:**

$$\begin{aligned}
&\Pr_{\overline{c}\in(\mathcal{Z}_p)^n,\ \overline{d}\in(\mathcal{Z}_p)^{n-1}}[\\
&g(\overline{a},\overline{b}') && = g(\overline{c},\overline{d}') + g(\overline{c},\overline{b}'-\overline{d}') + g(\overline{a}-\overline{c},\overline{d}') + g(\overline{a}-\overline{c},\overline{b}'-\overline{d}')\\
& && = g(\overline{c},\overline{d})\odot\overline{x} + g(\overline{c},\overline{b}-\overline{d})\odot\overline{x} + (\overline{a}-\overline{c},\overline{d})\odot\overline{x} + g(\overline{a}-\overline{c},\overline{b}-\overline{d})\odot\overline{x}\\
& && = g(\overline{c}+\overline{a}-\overline{c},\overline{d}+\overline{b}-\overline{d})\odot\overline{x}\\
& && = g(\overline{a},\overline{b})\odot\overline{x}] \qquad > 1-4\epsilon > 0
\end{aligned}$$

The first and third equalities hold from the linearity of $g$ with probability 1. The second one holds from the high-probability consistency of $g$ with respect to left shifts with probability $> 1-4\epsilon$. The last one is just a rewriting of the terms. Since they all hold with nonzero probability, the first term is equal to the last with probability 1. $\Box$

The linearity of $g$ can be shown using a similar proof to that in [BLR90]. These three properties uniquely define polynomial multiplication.

**Lemma 6** *If $g$ has the properties mentioned in the previous two lemmas, and is always linear, then $g$ has to be the polynomial multiplication function.*

**Proof:** It is given that $g(\overline{r},\overline{a}) = f(\overline{r},\overline{a})$ for all $\overline{a}$. The generators in this case are the vectors that have a 1 in one position and 0's everywhere else. $g$ is consistent with respect to the first one of these generators, namely $\overline{r}$. The rest can be obtained using of the fact that $g$ is consistent with respect to shifting left, to prove that $g(\overline{t},\overline{a}) = f(\overline{t},\overline{a})$ for any size $n$ vector $a$ and any generator $\overline{t}$. Then, it can be shown using the linearity property that $\overline{t}$ can be replaced with any size $n+1$ vector in this equatoin. $\Box$

**Theorem 7** *The above program is a $(0,\epsilon)$-tester for polynomial multiplication.*

**Proof:** The proof is very similar to that for the FFT. If $p = f$, then $p$ will clearly pass all of the tests. If $p(x) = f(x)$ for less than a $1-\epsilon$ fraction of the inputs $x$, then from lemma 6 and from the fact that $g$ is equal to $p$ at least $1-\delta$ of the time, $g$ cannot have all three of the properties mentioned in Lemma 6.It can be shown using Chernoff bounds that in that case the probability of $p$ passing all of the tests is at most $\delta$. $\Box$

# 6 Matrix Multiplication

**Definitions:** Let $f(A,B)$ denote the the product of two $n\times n$ matrices $A$ and $B$. Also, let $G_{i,j} \in (\mathcal{Z}_p)^{n\times n} denote the matrix that has a 1 in position$(i,j)4, and a ) everywhere else. We define:

$$g(A,B) \equiv \mathrm{maj}_{C,D\in(\mathcal{Z}_p)^{n\times n}}\{p(C,D)+p(A-C,D)+p(C,B-D)+p(A-C,B-D)\}.$$

Let $P$ be a permutation matrix: $\begin{bmatrix} 0 & 1 & 0 & \dots & 0\\ 0 & 0 & 1 & \dots & 0\\ \vdots & \vdots & \vdots & & \vdots\\ 1 & 0 & 0 & \dots & 0 \end{bmatrix}$

**The Test:**

In this example the generators are the $G_{i,j}$'s: matrices with a 1 in one place only and 0's everywhere else. The tests first ensure that $g$ is correct all the time when its second arguments is one of the generators $G_{1,i}$. Any generator can be obtained from any one of these $n$ generators by rotating it horizontally and/or vertically a certain number of times. Therefore, $p$ is tested to determine whether such rotations on its input have the desired effects. This is achieved by several tests: The first test checks whether $p$ is linear for most inputs. The second test determines whether $g$ is correct when its second input is one of the $n$ generators $G_{1,i}$. Tests 3 and 4 ascertain that $g$ gives the correct answer when either one of its inputs is the permutation matrix $P$. Any generator $g_{ij}$ can be obtained from another generator $G_{i',j'}$ by multiplying it with $P$ several times on both sides, since multiplication by $P$ rotates a matrix one row or one c! olumn depending on whether $P$ is

/*Test 1*/
$count$ = 0;
for $i = 1 \ldots m$ do
    pick random $A, B, C, D \in (\mathcal{Z}_p)^{n\times n}$
    if $p(A,B) + p(A,C) + p(B,C) + p(B,D) = p(A+B, C+D)$ then $count = count + 1$
if $count < q$ then FAIL

/*Test 2*/
count = 0;
subcount = 0
for $i = 1 \ldots m$ do
    pick random $C, A' \in (\mathcal{Z}_p)^{n\times n}$, zero out all but the first row of $A'$ to get $A$
    for $j = 1 \ldots l_2$ do
        pick random pairs $B, D \in (\mathcal{Z}_p)^{n\times n}$
        if $p(B,D) + p(C-B, D) + p(B, A-D) + p(C-B, A-D) = f(C, A)$
        then $subcount = subcount + 1$
if $subcount \geq q$ then $count = count + 1$ else FAIL
        if $count < q$ then FAIL.

/*Test 3*/
count = 0;
subcount = 0
for $i = 1 \ldots m$ do
    pick random $C \in (\mathcal{Z}_p)^{n\times n}$
    for $j = 1 \ldots l_2$ do
        pick random pairs $B, D \in (\mathcal{Z}_p)^{n\times n}$
        if $p(B,D) + p(C-B, D) + p(B, P-D) + p(C-B, P-D) = f(C, P)$
        then $subcount = subcount + 1$
if $subcount \geq q$ then $count = count + 1$ else FAIL
        if $count < q$ then FAIL.

/*Test 4*/
Repeat Test 3 with P as the first argument to $p$ and $f$

*Continued on next page*

*Continued from page 9*

```
/*Test 5*/
count = 0;
for i = 1...m do
    pick random X, Y, Z ∈ (Z_p)^{n×n}
     for j = ⌈1...log_2(5/ε)⌉ do
         pick random A, B, C, D ∈ (Z_p)^{n×n}
         temp_1 = p(A, B) + p(X − A, B) + p(A, Y − B) + p(X − A, Y − B)
         g1_j = p(C, D) + p(temp_1 − C, D) + p(C, Z − D) + p(temp_1 − C, Z − D)
         temp_2 = p(A, B) + p(Y − A, B) + p(A, Z − B) + p(Y − A, Z − B)
         g2_j = p(C, D) + p(A − C, D) + p(C, temp_1 − D) + p(A − C, temp_1 − D)
    if all of the g1_j's or g2_j's are not the same then FAIL
    if g1_1 = g2_1 then count = count + 1
if count ≥ ln(4/δ)(1 − ε/5) + 1/√2 then PASS else FAIL
```

It can be proven with methods similar to those in [BLR90] that if $p$ is linear for at least a $\geq 1-\delta/2$ fraction of its inputs, then $g$ is the same as $p$ on at least $1-\delta$ fraction of the inputs, and is always linear.

Test 2 tells us about $n$ of the generators:

**Lemma 8** *If $g(C,A) = f(C,A)$ for at least $1-\epsilon$ of the time, where $C \in (\mathcal{Z}_p)^{n\times n}$'s and $A$ is the 0-matrix except that its first row is a random size $n$ vector, then $g(C, G_{1,i}) = f(C, G_{1,i})$ for all $C$ and $i = 1, \ldots, n$.*

**Proof:** It can be shown that $g(C,A) = f(C,A)$ for all $C \in (\mathcal{Z}_p)^{n\times n}$ and all $A$ that are 0 everywhere but the first row, using the proof methods that are used throughout this paper to prove that certain properties of $g$ are true all the time if they are true a certain fraction of the time. Since $g$ is linear all the time, if there is a $G_{1,i}, D \in (\mathcal{Z}_p)^{n\times n}$ pair such that $g(G_{1,i}, D) \neq f(G_{1,i}, D)$, there must be at least one other generator $G_{1,j}$ such that $g(G_{1,j}, D) \neq f(G_{1,j}, D)$ to cancel the error due to the first incorrect computation so that $g(C,A)$ remains correct. However, an adversary can always pick such a matrix $C'$ that the errors from those generators do not cancel out. This results in $g(C', A)$ not being equal to $f(C', A)$, which contradicts what the test has established about $g$. $\square$

Next, we show that linearity, combined with the associativity result for $g$ from Test 5, establishes another property for $g$:

**Lemma 9** *If $\epsilon < 1/5$ and $g$ is always linear and associative for a $(1-\epsilon/2)$ fraction of the time, then $g$ is always associative.*

**Proof:**

$$\begin{array}{rcl} \Pr_{X_1,Y_1,Z_1}[g(g(X,Y),Z) & = & g(g(X_1,Y_1),Z_1) + g(g(X_1,Y_1),(Z-Z_1)) + \cdots \\ & = & g(X_1, g(Y_1,Z_1)) + \cdots \\ & = & g(X, g(Y,Z)] \qquad > 1-5\epsilon > 0 \end{array}$$

The first equality is true from the linearity of $g$ with probability 1. The second one is true by the result of Test 5 with probability $> 1/5$. The last one is just a rewriting of the matrices involved, thus, it is true with probability 1. Since the probability is strictly greater than 0, and the first and the last terms of the set of equalities do not mention $X_1, Y_1$ or $Z_1$, they must be equal with probability 1. Therefore $g$ is associative for all inputs. $\square$

It can also be shown using similar techniques that $g$ computes $f$ whenever one of its arguments is $P$.

Now it can be shown that the properties tested uniquely define matrix multiplication:

**Lemma 10** *If $g$ is always linear and associative, computes $f$ whenever either one of its arguments is $P$, or the first one is one of the $G_{1,i,1}$'s, or the second one is a $G_{i,1}$, then $g$ has to be the matrix multiplication function.*

**Proof:** ¿From linearity, we can write

$$g(X,Y) = a_{1,1,1,1}.g(G_{1,1}, G_{1,1}) + a_{1,2,1,1}.g(G_{1,2}, G_{1,1}) + \cdots$$

where $a_{i,j,k,l}$ is a scalar constant which is equal to $X_{i,j}.Y_{k,l}$.

If $g(G_{i,j}, G_{k,l}) = f(G_{i,j}, G_{k,l})$ for any $i$, $j$, $k$, $l$, then the linearity property implies that $g$ is the same as $f$.

$$\begin{aligned} g(G_{i,j}, G_{k,l}) &= g(f(G_{i,1}, P^{j-1}), f(P^{k-1}, G_{1,l})) \\ &= g(g(G_{i,1}, P^{j-1}), g(P^{k-1}, G_{1,l})) \\ &= g(G_{i,1}, G_{q,l}) \\ &= f(G_{i,1}, G_{q,l}) = f(G_{i,j}, G_{k,l}) \end{aligned}$$

$P^k$ denotes $f(f(f(\ldots(P,P)\ldots)))$, where the multiplication is applied $k-1$ times. The first equality is just a rewriting of the two generators in terms of other generators and a number of rotations. The second one follows from the fact that $g$ computes $f$ whenever one of its arguments is equal to $P$. The third one follows from associativity of $g$, and the second generator is yet another rewriting of the multiplication of several $P$'s and $G_{1,l}$. The fourth one is true because $g$ computes $f$ correctly when its first argument is $G_{i,1}$, and $q = k + j - 2$. The last one is a rewriting of the previous one, using the associativity of multiplication. Therefore, $g$ is the same function as $f$. $\square$

**Theorem 11** *The above program is a $(0, \epsilon)$-tester for matrix multiplication.*

**Proof:**

Since $g$ computes $f$ and is $(1-\epsilon)$-close to $p$, $p$ is $1-\epsilon$-close to $f$. If $p$ is not $\epsilon$ correct, then it will have to fail at least one of the tests with probability at least $1-\delta$. If it is always equal to $f$, it will pass all of them with probability 1.

$\square$

## References


[BF90] D. Beaver and J. Feigenbaum. Hiding instances in multioracle queries. In *Proceedings of the 7th Annual Symposium on Theoretical Aspects of Computer Science, Springer Verlag LNCS 415*, pp. 37– 48, 1990.

[BK89] M. Blum and S. Kannan. Program correctness checking ... and the design of programs that check their work. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pp. 86–97, 1989.

[BLR90] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pp. 73–83, 1990.

[Fre79] R. Freivalds. Fast probabilistic Algorithms. Springer Verlag Lecture Notes in CS No.74, Mathematical Foundations of CS, pp. 57-69, 1979.

[CL90] Cleve, R., Luby, M., "A Note on Self-Testing/Correcting Methods for Trigonometric Functions", International Computer Science Institute Technical Report TR-90-032, July, 1990.

[GLRSW91] P. Gemmell, R. Lipton, R. Rubinfeld, M. Sudan, and A. Wigderson. Self-testing/correcting for polynomials and for approximate functions. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pp. 32–42, 1991.

[Lip91] R. Lipton. New directions in testing. *Distributed Computing and Cryptography, DIMACS Series in Discrete Math and Theoretical Computer Science, American Mathematical Society*, 2:191–202, 1991.

[RS92] R. Rubinfeld and M. Sudan. Testing polynomial functions efficiently and over rational domains. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 23–43, 1992.

[RS93] R. Rubinfeld and M. Sudan. Robust Characterizations of Polynomials and their Applications to Program Testing. Available as IBM Research Report RC 19156 (83446) 9/9/93 and Cornell CS Tech. Report 93–1387, September 1993.