

A Guide to Self-Testing/Correcting Pairs: Everything you ever wanted to know and more

Casey Warmbrand

March 2003

Abstract

For any given function, f , we can create a **self-testing/correcting pair**, that will be able to test any program, P , that is supposed to compute f . If P passes the test, it can then be corrected. This paper will summarize the results of Blum, Luby, and Rubinfeld, from their paper “Self-Testing/COrrrecting with Applications to Numerical Problems.” We will show that for a given confidence parameter, β , both the tester and corrector can be accurate with probability $1 - \beta$, and that all this work can be done efficiently.

1 Introduction

Suppose we have a program P that computes a function f . How are we to know if P does this correctly? Classically two methods have been used to do this. One is called *program verification*, this is where a formal proof of correctness is done. The other classical method for answering the question of whether or not P is correct, is *traditional program testing*, also called *spot testing*. There are quite a few issues involved with both these methods, that we will address shortly. These problems are what form the basis for a Self-Testing/Correcting pair. A tester, for a given function, will return either Pass or Fail, depending upon whether P computes a high enough ratio of random inputs correctly, or not. Provided the program passes the test, the corrector can then be used, to almost certainly compute a given value of the function correctly. In other words, what the tester says is that P is not too faulty, and then the corrector is able to compute the correct value of a given

input, to within as high an accuracy as we would like. In order for this to be useful, it is necessary that both the tester and corrector be both *different* and *efficient*.

1.1 Some Basics

Consider the task of computing a given function f . To do this, we write a program P that will compute the values for us. So what is wrong with *program verification*? Well, for one thing, a proof of correctness, is nearly always far more complex than the program itself. Thus, for harder and harder functions, this becomes increasingly more difficult and possibly even impossible. So why not test inputs? In most cases, the domain, D of f will be quite large or even infinite. Thus, verifying that $P(x) = f(x)$ for all inputs $x \in D$ is not only difficult, in some cases, may even be impossible. Even if we do this, it only guarantees that this specific program P works, and doesn't even ensure that it will work correctly when compiled and run on possibly faulty hardware. And so, the need for a tester is established.

Not just any tester will work. First, we must ensure we test a wide enough range of inputs. Otherwise we can not be sure our program will work for the inputs we need to compute. Second, if we use some other program P' in order to test P , we are relying on P' to be correct. In many cases this is quite deadly; if a common mistake was made in coding P , it is likely the same mistake was made in coding P' , and our test will not be reliable. And so, Ronitt Rubinfeld, in her thesis and seminal paper with Manuel Blum and Michael Luby, have come up with a solution, *Self-Testing*. We will use the **same** program P on a sufficiently large, randomly selected range of inputs, to **test** whether or not P computes enough of these correctly to be deemed "good enough".

The idea is then to design a probabilistic program T_f that can self-test **any** program P that is supposed to compute f . T_f will make calls to P , to estimate the probability that $P(x) \neq f(x)$ for a random input x . If we run the tester on enough random inputs, and a high enough proportion of them pass the test, the tester will return PASS, thus assuring us that P is not too faulty. As long as we make sure that the running time of T_f is faster than that of any correct program for computing f , we can be sure that the tester must be doing something *quantifiably* different than computing f directly. We also want to ensure that T_f is *efficient*. This means, that we want the running time, including calls to P , to be within a constant multiplicative

factor of the running time of P . Otherwise, the advantage of this would be lost, due to a running time slowdown. These last two conditions ensure that our tester is both *different* and *efficient*.

Once a program passes its tester, we know that it computes most of its inputs correctly. That's good; however we must be sure that the input we want is one of the correct ones. This leads to the other half of the pair, the *self-corrector*. The idea here is to design a probabilistic program C_f that will be able to use P to compute $f(x)$ correctly, nearly all the time. Again we want C_f to be both *different* and *efficient*.

This self-testing/correcting pair, (T_f, C_f) , is a powerful tool. Since both the tester and corrector access P as an *oracle* or *black-box*, their results do not depend on what P is. This is a key fact in why self-testing/correcting is so important. Because of this, we can use this same testing/correcting pair, for **any** program that computes f . If some time in the future a faster program P' is created to compute f , the same pair can be used to ensure P' is not too faulty, and then to determine the correct value of $f(x)$ for a given x , using the new program P' . This reusability makes it quite worthy of our time, to create *self-testing/correcting* pairs, for many commonly used functions.

1.2 Formal Definitions

Some formal definitions, that will now be presented, will be useful in our study of these pairs.

We will consider functions of one input from a universe I . Let I_1, I_2, \dots be a sequence of subsets of I such that $I = \cup I_n$. Let $D = \{D_n | n \in \mathcal{N}\}$ be a collection of probability distributions such that D_n is a distribution on I_n . P is a program that supposedly computes f .

Let $error(P, f, D_n)$ be the probability that $P(x) \neq f(x)$ when x is randomly chosen in I_n according to D_n . Let $\beta > 0$ be a confidence parameter.

Definition: Let $0 \leq \epsilon_1 < \epsilon_2 \leq 1$. An (ϵ_1, ϵ_2) -*self-testing program* for f with respect to D is a probabilistic oracle program (makes calls to another program specified at run-time) T_f that has the following properties for any program P on input n and β .

1. If $error(P, f, D_n) \leq \epsilon_1$ then T_f outputs PASS with probability at least $1 - \beta$.
2. If $error(P, f, D_n) \geq \epsilon_2$ then T_f outputs FAIL with probability at least $1 - \beta$.

The value of ϵ_1 should be as close as possible to ϵ_2 to allow for as small a grey area as possible between the responses of “PASS” and “FAIL”. But there must be some gap, to ensure that we will still be able to differentiate between a program, P , that computes a large portion of the inputs correctly, and one that is close, but not good enough.

Definition: Let $0 \leq \epsilon < 1$. An ϵ -self-correcting program for f with respect to D is a probabilistic oracle program C_f that has the following property on input $n, x \in I_n$ and β . If $error(P, f, D_n) \leq \epsilon$ then $C_f(x) = f(x)$ with probability at least $1 - \beta$.

Definition: 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 a collection of distributions D such that T_f is an (ϵ_1, ϵ_2) -self-testing program for f with respect to D and C_f is an ϵ -self-correcting program for f with respect to D .

With the definitions and notation out of the way, let us take a look at an example, and see how all this can work.

2 A Brief Example: The mod Function

At this point it will be valuable to consider, probably, the simplest example of a self-tester/corrector that there is. This is for the mod function. That is, the function, $f(x, R) = x \text{ mod } R$. It will be easier to understand if I present the self-corrector first and the tester afterwards, plus the corrector is more intuitive.

We must first assume that the $error(P, f, U_Z \times U_R)$ is sufficiently small. The following is an ϵ -self-correcting program for f , making calls to an oracle program P , with confidence parameter β .

2.1 Programs for Mod

The following programs are variations of the algorithms presented in both [1] and [2]; the original code is given in the appendix.

Mod Self-Corrector (D, R, x, β)

```
generate random  $x_1 \in D$ 
let  $x_2 := x -_D x_1$ 
let  $ans_i := P(x_1, R) +_R P(x_2, R)$ 
Repeat this  $N$  times
return the most common  $ans_i$ 
```

The D above, represents some specified, large range, over which we will choose our random values that we use to correct $P(x, R)$. The range is usually set to be the integers *mod* $R2^n$, for some specified n . The notation used for $+_M$ and $-_M$, denote addition and subtraction *mod* M . However it is not really *mod* M , since these will never be less than $-M$ or greater than $2M$, so a simple comparison, and subtraction or addition, of a correcting factor of M can be used, rather than computing something *mod* M . This notation will be used for the remainder of the paper.

Lemma 1: The above program can represent an ϵ -corrector.

Proof: We assumed P , was mostly correct, say at least $4/5$ of the time. Thus, $P(x_i, R)$ will be incorrect only $1/5$ of the time. Since $x := x_1 +_D x_2$, $ans_i \neq f(x, R)$ whenever either $P(x_1, R)$ or $P(x_2, R)$ are incorrect, this happens with probability $2/5$. Thus more than half ($3/5$), of the ans_i 's should be correct. And so, by repeating this a sufficiently large number of times (N), we can ensure, with high confidence $(1-\beta)$, that our program returns $f(x, R)$.

We made a vital assumption, that was the key fact in showing that this works; P is mostly correct. How can we ensure this to be true? Well that is exactly what it means to say that P , passed the *self-test*.

Before we consider the tester, it will help us to stop and think about the mod function itself first. The mod function is a linear function, that is $f(x + y) = f(x) +_R f(y) \forall x, \forall y$. This is what we will base our test on. We will use two separate tests, the LinearTest and NeighborTest.

LinearTest(D, R, β)

```
generate random  $x_1$  and  $x_2 \in D$ 
let  $x := x_1 +_D x_2$ 
If  $P(x, R) \neq P(x_1, R) +_R P(x_2, R)$  then  $t \leftarrow t + 1$ 
Repeat this  $N$  times
If  $t/N \geq \epsilon_2$  return FAIL
If  $t/N \leq \epsilon_1$  return PASS
```

NeighborTest(D, R, β)

```
generate random  $z \in D$ 
let  $z' := z +_D 1$ 
If  $P(z', R) \neq P(z, r) +_R 1$  then  $t' \leftarrow t' + 1$ 
Repeat this  $N'$  times
If  $t'/N' \geq \epsilon_2$  return FAIL
If  $t'/N' \leq \epsilon_1$  return PASS
```

2.2 Correctness

In the Linear Test, since x_1 and x_2 are chosen randomly, and x is fixed based on these random choices, the verification done on the third line is justifiable. We are repeating this process enough times to get a good feel of how accurate P is at computing random inputs of the function. Each time P is incorrect, t increments. Therefore, after repeating N times, the ratio t/N represents the percent incorrect. This ratio (percentage), is then compared with the specified values of ϵ_1 and ϵ_2 , to determine whether the function passes the Linear Test or not. The Neighbor Test does nearly the same thing, except, here only one thing is randomly selected. And now we are checking to see how P computes it, and the number one bigger than it. Now, t'/N' represents the percentage incorrect. And we compare that to the same ϵ -values. In order for a program to pass the tester, it must pass both the Linear and Neighbor tests, if it fails either one, it fails the test.

Lemma 2: The above programs, LinearTest and NeighborTest, taken together, form an (ϵ_1, ϵ_2) -tester

Proof: The proof of why this works is far more complex than for the corrector. But it should be clear, that as long as a high percentage of the rounds of these tests come out correct, we will pass the tester; and, if enough rounds are done, we can be fairly certain, that with high probability P is mostly correct. This will follow later, when the general tester is presented.

A few more notes before we continue on to the general examples. This is an example of what we called an (ϵ_1, ϵ_2) -tester, for $\epsilon_2 > \epsilon_1$. And, for $\epsilon \geq \epsilon_2$, we have an ϵ -corrector (i.e. it passed the tester). Obviously, since everything is done randomly, these are not *always* correct. But we don't need them to be, we only need them to be correct with probability $1 - \beta$. Since β , is an arbitrary input, we need to show that we can get these to be as good as we want. This is easily done, since the value of N (our number of repetitions), determines how accurate the test is, and how accurate the corrector is at giving back $f(x, R)$. Thus, N must be chosen to be some function of the given input confidence parameter, β . The program, P , has some *error* associated with it (as defined earlier), call it α . And so, for our corrector, to be incorrect, we need more than half of the ans_i 's to be incorrect. Each ans_i , is incorrect with probability $2(1 - \alpha) = \lambda$, it follows that more than half of them will be incorrect with probability less than, roughly, $\lambda^{N/2}$. And we want to be wrong less than β of the time, so we need $\beta > \lambda^{N/2}$. Taking the log and using some crafty algebra, leads us to, $N = O(\log(1/\beta))$. The constant from the Big-O, will depend on λ , and lead us to the specific ϵ value for the corrector. A very similar analysis will show that the values for N in the tester, will also be $O(\log(1/\beta))$, and these constants will also depend on λ and lead us to the values of ϵ_1 and ϵ_2 . The code in the appendix appears with these constants, and the ϵ -values are given afterward.

3 The General Forms

Now that we have seen a concrete example of how these self-testing/correcting pairs can be made, let us take a look at their general forms. As before, for simplicity, we will look at a form of a Generic Self-Correcting program first, and then that of the more complex Generic Self-Testing program.

Both the self-tester and self-corrector about to be introduced, exploit the following property:

Random Self-Reducibility: Let x be the input, which we wish to correct, and let $c > 1$ be an integer. The property is that, instead of computing $f(x)$, which is hard, we can express $f(x)$ in terms of F , some other easily computable function. F takes inputs $x, a_1, \dots, a_c, f(a_1), \dots, f(a_c)$. The a_i 's must be easily computable and randomly distributed, given x .

In the above example of the Mod function, this property was used in both the tester and corrector. F , was based on the idea that $(x + y) \bmod R = (x \bmod R) +_R (y \bmod R)$.

This property then allows us to use a program which is correct on a large fraction of inputs, to compute *any* input correctly with high probability.

3.1 General Corrector

GenSelfCorrector(D, x, β)

```

    randomly generate  $a_1, \dots, a_c$  based on  $x$ 
    For  $i = 1, \dots, c$ 
         $\alpha_i \leftarrow P(a_i)$ 
     $ans_m \leftarrow F(x, a_1, \dots, a_c, \alpha_1, \dots, \alpha_c)$ 
    Repeat this  $N$  times
    return the most common  $ans_m$ 

```

Theorem 1: This general corrector can be adapted to an ϵ -corrector for many functions.

Proof: Similar to proof of Lemma 1. Assuming the *error* of P is less than ϵ , most of the α_i 's will be correct. During an iteration, if all α_i 's are correct, then F will compute $f(x)$ correctly. So, by repeating an appropriate number of times, the most common ans_m will be $f(x)$ with as high a probability as we like.

It should be clear that this is exactly the form that was used for the above program **Mod Self-Correct**. This type of self-correcting program can also be used for the functions integer multiplication, modular multiplication, modular exponentiation, matrix multiplication over a finite field, multiplication of polynomials over a finite field, and several other problems. In all cases, this correcting program is both *different* and *efficient*.

3.2 Generic Linear Self-Tester

In order to fully understand the way the Generic tester works, it is necessary to introduce a bit of group theory notation.

All groups are assumed to be abelian. G is a finite group with operation \circ and with generators g_1, \dots, g_c and identity element 0 . Let G' be a finite or countable group with group operation \circ' and identity element $0'$. Let $f : G \rightarrow G'$ be a function. f must be hard to compute in comparison to \circ or \circ' . Let U_G be the uniform distribution on G . f has the *linearity property* if:

- (1) Choosing a random element in G , according to U_G is easy.
- (2) F is an easily computable function, such that for any pair $x_1, x_2 \in G$, $F(x_1, x_2) \in G'$ and further $f(x_1 \circ x_2) = f(x_1) \circ' f(x_2) \circ' F(x_1, x_2)$. This is called *linear consistency*. In all examples except integer multiplication, $F(x_1, x_2) = 0'$ for all inputs, thus making f a group homomorphism.
- (3) For each generator $g_i \in G$, F_i is an easily computable function with the property that, $\forall z \in G$, $F_i(z) \in G'$ and further $f(z \circ g_i) = f(z) \circ' F_i(z)$. This is the *neighbor consistency* property. Again integer multiplication is an exception, where this is not needed. For all other applications, both G and G' are cyclic groups, thus generated by the single element 1 and $1'$, and $\forall z \in G$, $f(z \circ 1) = f(z) \circ' 1'$.

These properties look complex. However, in most examples, the function f is a natural homomorphism. This is just a general way of summing up the ideas used for the Linear and Neighbor tests, like that seen in the Mod example.

Generic Self-Tester(ϵ, c, β)

Run **Generic LinearTest** and **Generic NeighborTest**
If both PASS, PASS *else* FAIL

Generic LinearTest(ϵ, β)

randomly choose $x_1, x_2 \in G$ according to U_G
Do for $i = 1, \dots, c$
 If $P(x \circ g_i) \neq P(x_1) \circ P(x_2) \circ' F(x_1, x_2)$ then $t \leftarrow t + 1$ and
 exit loop
Repeat this N times
If $t/N \geq \epsilon_2$ **return** FAIL
If $t/N \leq \epsilon_1$ **return** PASS

Generic NeighborTest(c, β)

randomly choose $z \in G$ according to U_G
Do for $i = 1, \dots, c$
 If $P(z \circ g_i) \neq P(z) \circ' F_i(z)$ then $t' \leftarrow t' + 1$ and **exit** loop
Repeat this N' times
If $t'/N' \geq \epsilon_2$ **return** FAIL
If $t'/N' \leq \epsilon_1$ **return** PASS

Theorem 2: This general linear tester can be adapted to an (ϵ_1, ϵ_2) tester for many functions.

Proof: The authors of [1] define a *discrepancy* function, $disc(y) = f(y) \circ' P(y)^{-1}$. Thus, P computes f correctly if and only if this *disc* function is a homomorphism from G into $\{0'\}$. If G' is infinite with no finite subgroups, then ensuring that *disc* is a homomorphism is enough to force it to map to $\{0'\}$. If there are finite subgroups, the **NeighborTest** ensures that *disc* maps to $\{0'\}$. We do not need *disc* to map perfectly into $\{0'\}$, we just need it to mostly map into $\{0'\}$, that way we are confident that P is mostly correct. The authors provide some of the work, which draws results from

Probability theory on groups. The basic idea is to show that as long as the probabilities that $[disc(x_1 \circ x_2) \neq disc(x_1) \circ' disc(x_2)]$, $[disc(y) \neq 0']$ and $[disc(z) \neq disc(z \circ g_i)]$ are all small, when x_1, x_2, z are randomly chosen in G according to U_G , then we can show that $disc$ is close to the desired homomorphism. Thus, P computes most inputs correctly.

This form of a tester can be used for all the same functions that were listed for the General Corrector, except integer multiplication. The only slight modification necessary for integer multiplication, is to remove the NeighborTest, and just run the LinearTest. This is because the integers have no finite subgroups.

We will now examine a couple more examples of how these general programs can be adapted to commonly used functions.

4 Example: Modular Multiplication

For the remainder of the paper, the code provided in [2] will be used. Now that the general forms are understood, it will be helpful to see some sets of values for N and N' , that give way to specified ϵ -values.

The function we will be considering here, takes positive integers x, y , and R , define $f(x, y, R) = x \cdot_R y$ (x times $y \bmod R$). If we assume G and y to be fixed, f is a function of x . Let $G = Z_{R^{2^n}}$ where \circ is $+_{R^{2^n}}$ and Let $G' = Z_R$ where \circ' is $+_R$. The key to the self-tester is the fact that, for any pair $x_1, x_2 \in Z_{R^{2^n}}$, $f(x_1, y, R) +_R f(x_2, y, R) = f((x_1 +_{R^{2^n}} x_2), y, R)$. Thus $F(x_1, x_2, y) = 0'$. If we use $\epsilon = 1/16$, the following is a $(1/864, 1/16)$ -self-testing program for f with respect to $U_{Z_{r^{2^n}}} \times U_{Z_{r^{2^n}}} \times U_R$. The input is n, R and the confidence parameter β .

Program ModMultSelfTest(n, R, β)

```

 $N = 1152 \ln(4/\beta)$ 
 $total \leftarrow 0$ 
Do for  $m = 1, \dots, N$ 
    Call ModMultLinearConsistency( $n, R, answer$ )
     $total \leftarrow total + answer$ 
If  $total/N > 1/144$  then Return FAIL

```

$N' = 32 \ln(4/\beta)$
 $total' \leftarrow 0$
 Do for $m = 1, \dots, N'$
 Call **ModMultNeighborConsistency**($n, R, answer$)
 $total' \leftarrow total' + answer$
 If $total'/N' > 1/4$ then Return FAIL else Return PASS

ModMultLinearConsistency($n, R, answer$)

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

ModMultNeighborConsistency($n, R, answer$)

Choose $y \in_U Z_{R2^n}$
 Choose $z \in_U Z_{R2^n}$
 $z' \leftarrow z +_{R2^n} 1$
 If $P(z, y, R) +_R y = P(z', y, R)$ then $answer \leftarrow 0$ else $answer \leftarrow 1$

Now suppose the $error(P, f, U_{\times} U_{Z_{R2^n}}) \leq 1/16$ (i.e. P passed the tester above), the following program is a $1/16$ -self-correcting program for f . The input is n, x, y , and the confidence parameter β .

ModMultSelfCorrect(R, x, y, β)

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

RandomSplit is defined in the appendix. This works because of the way modular multiplication works. Because $x = x_1 + x_2 - cR2^n$ and $y = y_1 + y_2 - dR2^n$, $x \cdot_R y = (x_1 \cdot_R y_1) +_R (x_1 \cdot_R y_2) +_R (x_2 \cdot_R y_1) +_R (x_2 \cdot_R y_2)$ (note: $cR2^n$ and $dR2^n \bmod R$ are both 0), thus the most common answer will be $(x \cdot_R y)$ with probability $1 - \beta$.

5 Example: Integer Multiplication

As mentioned before, this doesn't behave exactly like the previous two functions presented. The reason for this is that we are no longer dealing with a finite field. The function we are concerned with here is $f(x, y) = x \cdot y$, where x and y are positive integers. As mentioned before, there will only be a linear consistency test, the neighbor test is omitted. Again we will fix y to an arbitrary value and deal with f as a function of x , with $G = Z_{2^n}$ where \circ is $+_{2^n}$ and $G' = Z$ where \circ' is $+$. For $x_1, x_2 \in Z_{2^n}$, let $c = 1$ if $x_1 + x_2 \geq 2^n$ and let $c = 0$ otherwise, and let $x = x_1 + x_2 - c2^n = x_1 +_{2^n} x_2$ (since $c2^n = 0 \bmod 2^n$). We use the fact that $f(x_1, y) + f(x_2, y) = f(x, y) + y2^n$. Its important to see that $F(x_1, x_2) = yc2^n$ is easily computable. The following (modified slightly from the generic tester) will be a $(1/864, 1/16)$ -self-testing program for f . The inputs are n and a confidence parameter β .

Program IntegerMultSelfTest (n, β)

```

 $N = 1152 \ln(2/\beta)$ 
 $total \leftarrow 0$ 
Do for  $m = 1, \dots, N$ 
    Call IntMultLinearConsistency $(n, answer)$ 
     $total \leftarrow total + answer$ 
If  $total/N > 1/144$  then Return FAIL else Return PASS

```

IntMultLinearConsistency $(n, answer)$

```

Choose  $y \in_U Z_{2^n}$ 
Choose  $x_1 \in_U Z_{2^n}$ 
Choose  $x_2 \in_U Z_{2^n}$ 
 $x \leftarrow x_1 +_{2^n} x_2$ 
 $c \leftarrow (x_1 + x_2) \text{ div } 2^n$ 

```

If $P(x_1, y) + P(x_2, y) = P(x, y) + cy2^n$ then $answer \leftarrow 0$
 else $answer \leftarrow 1$

Assuming our program passes the above test, $error(P, f, U_{Z^{2^n}} \times U_{Z^{2^n}}) \leq 1/16$. The following will be a $1/16$ -self-correcting program with respect to f and input n , and confidence parameter β .

Program IntegerMultSelfTest(n, x, y, β)

```

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

```

The calls to **RandomSplit** set $x = x_1 + x_2 - c2^n$ and $y = y_1 + y_2 - d2^n$, therefore $x \cdot y = x_1 \cdot y_1 + x_1 \cdot y_2 + x_2 \cdot y_1 + x_2 \cdot y_2 - cy2^n - dx2^n - cd2^n$. Thus, since each call to P is incorrect with probability at most $1/16$, the four calls each time through the loop will all be correct with probability at least $3/4$. If all four calls are correct $answer_m = x \cdot y$, since this should be the most common $answer_m$, the corrector will return $f(x, y)$ with probability at least $1 - \beta$.

6 Appendix

Here are some of the original lines of code given in [1] and [2] and some brief explanations.

6.1 Mod Function

We assume we have a program P , that computes $f(x, R) = x \bmod R$, such that $error(P, f, U_Z \times U_R) \leq 1/8$. The following is a $1/8$ -self-correcting program for f , making calls to an oracle program P , with confidence parameter β

Program Mod Self-Correct(n, R, x, β).

```
 $N \leftarrow 12 \ln(1/\beta)$   
Do for  $m = 1, \dots, N$   
    Call RandomSplit ( $R2^n, x, x_1, x_2, c$ )  
     $answer_m \leftarrow P(x_1, R) + P(x_2, R)$   
Return the most common answer in  $\{answer_m : m = 1, \dots, N\}$ 
```

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

```
Choose  $z_1 \in_U Z_M$   
If  $z_1 \leq z$  then  $e \leftarrow 0$  else  $e \leftarrow 1$   
 $z_2 \leftarrow eM + z - z_1$ 
```

The function **RandomSplit** chooses a random x_1 in the probability distribution and then fixes x_2 , so that their sum is x . The key to this working correctly, is in the fourth line of **Program Mod Self-Correct**. Since the error of the program is relatively small (less than $1/8$), we know that the probability of any randomly selected x being computed correctly by P is at least $7/8$. Thus in the fourth line of the program, $P(x_1, R)$ and $P(x_2, R)$ are both correct with probability at least $49/64$. When both of these are correct, $answer_m$ will be given the value of $f(x)$. And, since they are both correct more than half the time, almost certainly the most common $answer_m$ will be $f(x)$. The value of N is chosen so that the probability of being incorrect is reduced below β .

Program Mod Self-Test(n, R, β)

```
 $N \leftarrow 864 \ln(4/\beta)$   
 $t \leftarrow 0$   
Do for  $m = 1, \dots, N$   
    Call ModLinearTest ( $n, R, answer$ )  
     $t \leftarrow t + answer$   
If  $t/N > 1/72$  then Return FAIL
```

$N' = 32 \ln(4/\beta)$
 $t' \leftarrow 0$
 Do for $m = 1, \dots, N'$
 Call **ModNeighborTest** ($n, R, answer$)
 $t' \leftarrow t' + answer$
 If $t'/N' > 1/4$ then Return FAIL else Return PASS

Function ModLinearTest($n, R, answer$)

Choose $x_1 \in_U Z_{R2^n}$
 Choose $x_2 \in_U Z_{R2^n}$
 $x \leftarrow x_1 +_{R2^n} x_2$
 If $P(x_1, R) +_R P(x_2, R) = P(x, R)$ then answer $\leftarrow 0$ else answer $\leftarrow 1$

Function ModNeighborTest($n, R, answer$)

Choose $z \in_U Z_{R2^n}$
 $z' \leftarrow z +_{R2^n} 1$
 If $P(z, R) +_R 1 = P(z', R)$ then answer $\leftarrow 0$ else answer $\leftarrow 1$

This is a $(1/432, 1/8)$ -self-tester. The range, Z_{R2^n} is used so that the randomly chosen x 's and z 's have a wide range to be chosen from, but not infinite, otherwise the program couldn't work. Again, as in the corrector, the N is chosen to go along with β , and so the desired ϵ -values are attained.

6.2 Self-Reducibility

Random Self-Reducibility: Let $x \in I_n$. Let $c > 1$ be an integer. The property is that $f(x)$ can be expressed as an easily computable function F of x, a_z, \dots, a_c and $f(a_z), \dots, f(a_c)$, where a_1, \dots, a_c are easily computable given x and each a_i is randomly distributed in I_n according to D_n .

This property allows us to transform a program that is correct on a large fraction of inputs into a program that computes $f(x)$ correctly with high probability for *any* input x . The following is a $1/4c$ -self-correcting program for f with respect to D_n . The input is $n, x \in I_n$ and a confidence parameter β .

6.3 Generic Corrector

GenSelfCorrector(n, x, β)

```
Do for  $m = 1, \dots, 12 \ln(1/\beta)$ 
  randomly generate  $a_1, \dots, a_c$  based on  $x$ 
  For  $i = 1, \dots, c$ 
     $\alpha_i \leftarrow P(a_i)$ 
   $ans_m \leftarrow F(x, a_1, \dots, a_c, \alpha_1, \dots, \alpha_c)$ 
Return the most common answer in  $\{ans_m : m = 1, \dots, N\}$ 
```

Lemma 3: This is a $1/4c$ -self-correcting program for f with respect to D_n .

Proof: The $error(P, f, D_n) \leq 1/4c$ and for each $k = 1, \dots, c$, a_k is randomly distributed in I_n according to D_n . Thus all c outputs of P are correct with probability at least $3/4$ each time through the loop. If all c outputs are correct, then by random self-reducibility, $ans_m = f(x)$. And so, after $12 \ln(1/\beta)$ executions, the majority of the answers are equal to $f(x)$ with probability at least $1 - \beta$.

6.4 Generic Self-Tester

The following is an $(\epsilon/36, \epsilon)$ -self-testing program for f with respect to U_G for all G' (except infinite groups with no finite subsets other than $\{0\}$). The input is ϵ and a confidence parameter β .

Gen Self-Testing Program(ϵ, β)

```
 $N \leftarrow \lceil 72/\epsilon \ln(2/\beta) \rceil$ 
 $t \leftarrow 0$ 
Do for  $m = 1, \dots, N$ 
  Call GenLinearTest( $ans$ )
   $t \leftarrow t + ans$ 
If  $t/N > \epsilon/9$  then Return FAIL
```

```

 $N' \leftarrow \lceil 32 \ln(4c/\beta) \rceil$ 
Do for  $m = 1, \dots, N'$ 
   $ans \leftarrow 0$ 
  Do for  $i = 1, \dots, c$ 
    Call GenNeighborTest( $i, ans$ )
   $t' \leftarrow t' + ans$ 
If  $t'/N' > 1/4$  then Return FAIL else Return PASS

```

GenLinearTest(ans)

```

randomly choose  $x_1 \in G$  according to  $U_G$ 
randomly choose  $x_2 \in G$  according to  $U_G$ 
If  $P(x_1 \circ x_2) \neq P(x_1) \circ' P(x_2) \circ' F(x_1, x_2)$  then  $ans \leftarrow 1$  else  $ans \leftarrow 0$ 

```

GenNeighborTest(i, ans)

```

randomly choose  $x \in G$  according to  $U_G$ 
If  $P(x \circ g_i) \neq P(x) \circ' F_i(z)$  then  $ans \leftarrow 1$ 

```

This form of a tester can be used for all the functions listed previously for the corrector, except for integer multiplication. The only thing necessary to make this work for integer multiplication is the omission of the second half of the code (all parts concerning the neighborhood test). This tester is an $(\epsilon/36, \epsilon)$ -self-tester. A proof is provided both in [1] and [2].

7 References

- [1] Blum, M., Luby, M., Rubinfeld, R., “Self-Testing/Correcting with Applications to Numerical Problems”
- [2] Rubinfeld, R., “A Mathematical Theory of Self-Checking, Self-Testing, Self-Correcting Programs”