A Mathematical Theory of Self-Checking, Self-Testing and Self-Correcting Programs

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. Rather than 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) require only a small multiplicative overhead (usually constant) over the running time of P. A *program result checker* for f (as introduced by Manuel Blum) allows us to check that on particular input x, $P(x) = f(x)$.

We present general techniques for constructing simple to program self-testing/correcting pairs for a variety of numerical functions. The self-testing/correcting pairs introduced for many of the problems are based on the property that the solution to a particular instance of the problem can be expressed as the solution to a few random instances of the same size. An important idea is to design self-testing/correcting pairs for an entire *library* of functions rather than for each function individually. We extend these notions and some of the general techniques to check programs for some specific functions which are only intended to give good *approximations* to $f(x)$.

We extend the above models and techniques of program result checking and self-testing/correcting to the case where the behavior of the program is modelled as being adaptive, i.e. the program may not always give the same answer on a particular input. These stronger checkers provide multi-prover interactive proofs for these problems.

The theory of checking is also extended to parallel programs [Rubinfeld]. We construct parallel checkers for many basic problems in parallel computation.

We show that for some problems, result checkers which are much more efficient can be constructed if the answers are checked in *batches*, i.e. many answers are checked at the same time. For these problems, the multiplicative overhead of checking the result can be made arbitrarily small.

This research was supported by NSF grant CCR 88-13632, by the International Computer Science Institute and by an IBM Graduate Fellowship.

this is it!

Dedicated to my family
Jack, Rivka and Ilan Rubinfeld

Contents

1 Introduction **1**

2 Program Result Checking **9**

2.1 The Model 9

2.1.1 Correctness of Result Checker 11

2.1.2 Efficiency 12

2.2 Examples 13

2.2.1 Sorting 13

2.2.2 Matrix Multiplication 14

3 Self-Testing/Correcting Programs **15**

3.1 Related Work 17

3.2 The Basics 18

3.3 Self-Correcting 21

3.3.1 The Mod Function 21

3.3.2 Generic Self-Correcting Program 22

3.3.3 Integer Multiplication 23

3.3.4 Modular Multiplication 23

3.3.5 Modular Exponentiation 24

3.3.6 Integer Division 25

3.3.7 Matrix Multiplication 26

3.3.8 Polynomial Multiplication 26

3.3.9 Multivariate Polynomial Function 27

3.4 Linearity and Self-Testing 28
3.4.1 Mod Function 29
3.4.2 Generic Linear Self-Testing 30
3.4.3 Integer Multiplication 37
3.4.4 Modular Multiplication 38
3.4.5 Modular Exponentiation 39
3.4.6 Integer Division 41
3.5 Self-Testing Polynomial Functions 42
3.6 Bootstrap Self-Testing 46
3.6.1 Matrix Multiplication 48
3.6.2 Polynomial Multiplication 49
3.6.3 Modular Inverse 49
3.6.4 Modular Exponentiation 51

4 Approximate Result Checking and Self-Testing/Correcting 55
4.1 Approximate Self-Correcting 57
4.1.1 Quotient 58
4.1.2 Generic Approximate Self-Correcting Program 59
4.2 Approximate Linearity and Approximate Self-Testing 60
4.2.1 Quotient Function 61
4.2.2 Generic Approximately Linear Self-Testing 62
4.3 Open Question 65

5 Libraries and Linear Algebra 66
5.1 Definitions 67
5.2 The Linear Algebra Library 68
5.2.1 Matrix Multiplication 69
5.2.2 Matrix Inversion 70
5.2.3 Determinant 71
5.2.4 Matrix Rank 72

6 Result Checkers for Parallel Programs 75

6.1 The Parallel Program Result Checking Model 76

6.2 Simple Parallel Result Checkers 77

6.3 Computability by Random Inputs 78

6.3.1 Any Symmetric Function on n Bits 78

6.3.2 Special Symmetric Functions 79

6.3.3 Randomly Self-Reducible, Linear and Smaller Self-Reducible Problems 80

6.4 Consistency 81

6.4.1 Problems that can be solved using Dynamic Programming 82

6.4.2 All Pairs Shortest Path 83

6.5 Duality 84

6.6 Constant Depth Reducible Functions 85

7 Adaptive Programs and Cryptographic Settings 86

7.1 Related Work 89

7.2 Private/Adaptive Checker 90

7.3 Open Questions 93

8 Batch Result Checking 94

9 Conclusions 97

Chapter 1

Introduction

In his book, *Memoirs of a Computer Pioneer* [65], Maurice Wilkes writes :

> By June 1949 people had begun to realize that it was not so easy to get a program right as had at one time appeared. I well remember when this realization first came on me with full force. The EDSAC was on the top floor of the building and the tape-punching and editing equipment one floor below on a gallery that ran round the room in which the differential analyser was installed. I was trying to get working my first non-trivial program, which was one for the numerical integration of Airy's differential equation. It was on one of my journeys between the EDSAC room and the punching equipment that "hesitating at the angles of stairs" the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs. Turing had evidently realized this too, for he spoke at the conference on "checking a large routine".

His prediction that software faults would be one of the most crucial issues for programmers proved correct, and the issue of software reliability has been plaguing computer programmers since the earliest days of the field. In many cases, these bugs can have serious implications. [1]

More is known now than in 1949 about programming and ways to minimize errors in programs, yet the problem still flourishes because programs are being developed to solve

[1] Examples from recent history range from severe flooding to millions of dollars of losses for major phone companies: In 1983, severe flooding along the Colorado River that killed six persons was attributed to a "monumental mistake" in federal computer projections of snow-melt runoff. The underestimation of the runoff caused officials to dam up too much water before the spring thaw. On January 15,1990, a bug in the newly installed software at AT&T temporarily shut down half of the telephone system with hundreds of thousands of callers being unable to call or receive their long distance phone calls.

more complicated problems, and need to be clever in order to maximize efficiency and to take advantage of more complicated computer architectures.

Traditional Approaches to Software Reliability The problem of software reliability has been explored in the past with several classical approaches. As each of these approaches has its own strengths and weaknesses, they should be viewed as supplementary, rather than competing approaches. We describe two approaches of interest.

The first method, *program verification*, involves a formal proof of correctness of the program code which is done only once, and before the program is ever used. Thus, using it does not affect the running time of using the program. Though programs for many problems have been formally verified, these programs are usually quite simple and often much less efficient than other programs for the same problem. Relatively simple data structure routines are very difficult to verify. Even when a program is proven correct, the proof is often much longer and more complicated than the program itself, and thus is at least as susceptible to errors. Since verification is too tedious of a task to do by hand, efforts have been aimed at automating or semi-automating the process. Such efforts have met with limited success. Additional complexity is introduced when programming on parallel processors and distributed computing environments. These are among the reasons that [27, De Millo Lipton Perlis] use to argue that program verification should not be the only way of approaching the problem of software reliability. Furthermore, the proof of correctness only makes a statement about the program as it is written on paper, not as it is typed into the file, nor the compiled code generated from the file, nor the physical representation of the code loaded into the hardware where the program is to be executed.

The second method, traditional *program testing*, involves testing the program's correctness on some inputs. Traditional testing consists of generating random instances or special instances of the problem and determining whether the program is correct on those instances. Testing overcomes some of the problems of program verification, e.g. the testing is of the actual physical representation of the code loaded into the hardware. Testing is also normally done in a preprocessing stage, and does not affect the run time of the program. Since it is unreasonable to test the program on *each* input, the programmer is never certain that the program is correct on the particular inputs for which the correct result is needed. In addition, proper testing raises the following issues which must be addressed:

1. Which input distribution should the program be tested on? In many cases, the input distribution that will occur in practice is unknown and is not static.
2. What constitutes an acceptable test? Typically, either (1) the instances checked are generated so that they are small enough or special enough to be checked by hand, which is insufficient to test the general behavior of the program, or (2) an "independent" program is used to calculate the answer. The "independent" program

is often a previous version of the same program that is being replaced, or one that is written by different people (who are likely to make the same types of errors) and thus is not independent at all.

Program Result Checking Recently, Manuel Blum in [15, Blum] has proposed a promising new framework, *Program Result Checking*, for allowing progammers to build *result checkers* into their programs. People realize the importance of checking their own hand calculations, and have developed ad-hoc methods for doing so: for example, when solving a system of equations, people are taught to plug the solution back into the equations to verify the correctness of the result. In fact, many programmers put checks into their programs to verify that the results being computed are correct or are at least feasible solutions. Program Result Checking involves writing a result checker program, to be run in conjunction with the original program, that checks the work of the program, i.e. it obtains a *proof of correctness* of the result of the program on the *specific* input that the program was run on. The result checker does not then verify whether or not the program is correct, and in fact is not allowed to look at the program code: it simply verifies whether the program gives the correct answer on the inputs on which it is called.

As in algorithm design, where a different algorithm is designed for each function, the result checker is written specifically for the function that it is supposed to check. However, in algorithm design, several techniques have been developed which can be used to write algorithms for large classes of problems. Similarly, one of the partially fulfilled goals of the area of result checking is to find techniques which can be used to write result checkers for large classes of problems.

This approach allows one to use programs written by other people without having to trust them: since the result checker for a function is usually easier to write than the program for the function, it is easier to use the untrusted program in conjunction with the result checker than to write a new program for the function. Moreover, one can use programs that are based on unproven heuristics for the problem. For example, many good heuristics exist for the graph isomorphism problem. If a heuristic program is used, then the result checker either finds a mistake in the heuristic or else it is very likely that the heuristic has output the correct answer on the given input.

This framework is meant to address computational problems that have well-defined input/output specifications, rather than problems where the difficulty comes in deciding on the correct specifications. The result checker is written specifically for the function which it checks, but must work for all programs that purport to compute that function. The result checker may call the program on other inputs, but it may only access the program as a black box, and may not look at the program code. Thus, result checking is inherently different than program verification. Result checking is also of a different nature than verifying that the program is following an algorithm's steps correctly. However, we will see (Chapter 6) that some result checkers are in some sense using the program to reconstruct the steps of a very simple algorithm for the problem, to be used as a proof of the correctness of the result. If this can be done efficiently, then it becomes worthwile to use the faster but more complicated algorithm rather than the very simple but slow algorithm. We define a program result checker more formally in Chapter 2.

The first program result checkers for a number of important problems are given in [15, Blum], [17, Blum Kannan], [23, Blum Raghavan] [39, Kannan]. Program Result Checking has been successfully applied to a wide range of problems, including sorting, matrix rank, linear programming, graph isomorphism, matrix multiplication, greatest common divisor [2, Adleman Huang Kompella], [15, Blum], [17, Blum Kannan], [18, Blum Kannan Rubinfeld], [20, Blum Luby Rubinfeld 2], [23, Blum Raghavan], [34, Freivalds],[39, Kannan].

The question remains of how to verify that the result checker program 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 result checkers that have been 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 result checker to try and ensure that it is correct, because a result checker can be used on all revisions in the future to the currently used program for computing the function. Rather than trading the assumption that the program is correct for the assumption that the result checker is correct, [15, Blum] suggests that the result checker should be in some quantifiable way "different" than any program that correctly computes the function directly, because then it is unlikely that the result checker makes mistakes of the same type as those made by the program: specifically, [15, Blum] suggests the requirement that the running time of the result checker (not including the running time of the program on any of the calls that the result checker makes to it) be strictly faster than the fastest program for computing the function. Thus, the result checker must do something other than compute the function directly.

Since result checking is done at run time, care must be taken so that it does not significantly add to the total computation done. For many functions, the result checking task is significantly easier than the task of computing the function value. We say that a result checker is *efficient* if the total running time of the result checker, including the running time of the calls to the program, is linear in the running time of the program. We would like the result checkers to be efficient, or as close to efficient as possible.

The result checking is done with respect to the program being executed in the hardware. If the hardware is assumed to be non-faulty, no further assumptions are needed. If the hardware is assumed to be possibly faulty, then we assume that the simple result checker is loaded into a smaller non-faulty portion of hardware (similar to what is done with the operating system kernel). In this case, the result checker can even be used to find faults in the program due to hardware errors that develop over time.

An advantage of the approach of [15, Blum] is illustrated by the following example.

Suppose we have a correct program which we want to run as fast as possible. We have two kinds of hardware, non-faulty hardware and faster but possibly faulty hardware. There are two possibilities: (1) run the program alone on the non-faulty hardware; (2) run the result checker on the non-faulty hardware in conjunction with the program on the faster hardware. Because the result checking time is much smaller than the running time of the program when run on equal speed hardware, the bottleneck in running time in (2) is still the program. Thus, (2) can yield an overall gain in processing speed over (1) without compromising confidence in the correctness of the output.

Self-Testing/Correcting Programs We introduce the notion of *self-testing/correcting programs* (Chapter 3), which is an extension of the theory of program result checkers:

Although a result checker can be used to verify whether the program is correct on a particular input, it does not give a method for computing the correct answer in the case that the program is found to be faulty. We show that for many functions, faulty programs which are nevertheless correct on a substantial fraction of the inputs from a conveniently chosen distribution (determined by the function), can be *self-corrected*, i.e. turned into programs that are always correct. For example, we show how to convert a program for integer multiplication that errs on up to 1/8 of the pairs of n-bit inputs, into a probabilistic program that is correct on *every input* with high probability. Self-correcting has implications for program design: It may be easier to write programs that are allowed to err infrequently, because special cases can be ignored. As with result checking, we ask that the self-correctors be quantifiably different than any program that computes the function.

We discuss a property of functions which allows them to be self-corrected, and give a general technique for constructing self-correcting programs for all functions which have this property. Examples of functions which have this property include integer multiplication, integer division, the mod function, matrix multiplication, polynomial multiplication, modular multiplication, modular exponentiation and the evaluation of any function which is a multinomial.

In order to ascertain that the programs are usually right on the distribution chosen for the self-corrector, we introduce the notion of *self-testing programs.* The design of the self-corrector dictates which input distribution must be tested. We ask that the self-testers be be quantifiably different than any program that computes the function, which does not allow the tester rely on the existence of a "correct" program for the function in order to test the program. The theory of result checking gives one way of satisfying this requirement: one calls the program on the test input, and then uses the result checker to verify that the program gives the correct answer. The assertion is true because of the restriction that the result checker is quantifiably different than any program for the function, and therefore does not check the result by running another program for the problem to see if it gets the same answer. We give other techniques which can be used to construct very simple and efficient self-testers, without the use of a result checker for the function, for all of the functions mentioned above.

A self-testing/correcting pair is a powerful tool. A user can take *any* program that purportedly computes the function and self-test it. If the program passes the self-test then, on any input, the user can call the self-corrector, which in turn makes calls to the program, to correctly compute the function. Even a program that computes the function incorrectly for a small but significant fraction of the inputs can be used with confidence to correctly compute the function for any input. Furthermore, since either a fault is found in the program during the testing phase, or the correct value of the function is computed, the self-testing/correcting pair can be used to construct a result checker for the same function.

It is clear that self-correctors must be efficient (as was defined for result checkers), because they are used at runtime. We will see that in many cases, testing must also be done at runtime rather than in preprocessing, and therefore, the efficiency of the testing program is important.

If in the future somebody designs a faster program for computing the function then the same pair can be used to self-test/correct the new program 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.

The theory of self-testing leads to interesting mathematical questions about properties that characterize a function. It is shown that certain properties that characterize a function which hold *everywhere* can be replaced by the same property which only holds *almost everywhere*. For example, suppose f is a function that maps a group G to a group H. We say that f is linear if it has the property that for *all* x, y, $f(x +_G y) = f(x) +_H f(y)$ (where $+_G, +_H$ are the group operations over the domain and range respectively). The results in Chapter 3 relax the condition required for linearity in the following sense: they show that if for *most* 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. A similar property and relaxation is shown to hold for polynomials. Since it is computationally much easier to determine whether a property is satisfied most of the time than to determine whether it is always satisfied, this relaxation is important for self-testing.

Programs that Approximate In the notions of a result checker and self-testing/correcting pair considered so far, a result of a program is considered incorrect if it is not *exactly* equal to the function value. In Chapter 4, we extend the notions to *approximate* result checkers and *approximate* self-tester/correctors which check whether or not the program gives a good approximation (defined in terms of absolute error) to the exact value of the function. We give techniques which can be used to construct approximate result checkers/self-testers/correctors for programs that approximate the integer multiplication and integer division functions. The techniques also give approximate result checkers/self-testers/correctors for the quotient function, for which no exact self-tester/corrector or result checker is known.

Libraries Often programs for related problems 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. In Chapter 5, we extend the theory proposed in [15, Blum] to allow the use of several programs, or a *library*, to aid in result checking, testing and correcting. We show that this allows one to construct result checkers and self-testing/correcting pairs for functions which did not previously have efficient result checkers, self-testing programs or self-correcting programs. The result checker and 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 check a determinant program in the absence of programs for matrix multiplication and inverse. 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. 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.

Result Checkers for Parallel Programs A user is unlikely to be willing to use a sequential result checker to verify the correctness of a result produced by a fast parallel algorithm. In Chapter 6, we extend the program result checking framework to the setting of checking parallel programs and find general techniques for designing such result checkers. For example, we find techniques for result checking programs which compute certain types of functions that have the property that they can be computed be computed "indirectly", by calling the program on another, related input. We also present a techniques based on quickly reconstructing the computation of a simple sequential algorithm, on duality and on constant depth reducibility among problems. We find result checkers for many basic problems in parallel computation and show that for many problems, checking the parallel program's answer does not increase the parallel computation time and total work done by much.

Adaptive Checking and Interactive Proofs In the theory of program result checking introduced in [15, Blum], the program is always assumed to be a fixed program, whose output on any particular input is always the same. This is not always the case, as there are programs whose behavior changes as they run, even though the functions that they supposedly compute remain fixed. For example, hardware errors may evolve over time depending on the previous inputs that the program has been run on, or, the software may be written such that running the program on certain inputs may have unintended side effects on the program's future behavior. We show how to replace this assumption by the assumption that there are two copies of the program which do not affect each other in Chapter 7. In [15, Blum] [17, Blum Kannan], the relationship between result checkers and interactive proofs [37, Goldwasser Micali Rackoff] is studied. We discuss the relationship between result checkers and the multi-prover interactive proofs of [13, Ben-Or Goldwasser Kilian Wigderson].

Batch Result Checking Though many programmers are willing to spend some time overhead in order to verify that their programs give correct answers, for some applications, where efficiency is crucial, even a constant multiplicative time overhead makes result checking undesirable. In Chapter 8, we define a variant model of result checking, called *batch result checking*: Often greater efficiency can be achieved if the user does not need to know immediately whether the program gives the correct result. In this case, the result checker can wait until the program has been called on several inputs and check that the program is correct on all of the inputs at once. Batch result checking can allow greater efficiency, and we give examples of functions for which batch result checking allows one to reduce the overhead of the result checking process to the point where it is arbitrarily small.

Chapter 2

Program Result Checking

In this chapter, we describe the program result checking model proposed in [15, Blum]. We discuss several aspects of the model, and then we give examples of program result checkers.

2.1 The Model

DEFINITION 2.1.1 (probabilistic oracle program and running time) *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. The* incremental time *of M^A is the maximum over all inputs x of length n of the running time of $M^A(x)$, not counting the time for calls to A. The* total time *of M^A is the maximum over all inputs x of length n of the running time of $M^A(x)$, counting the time for calls to A.*

DEFINITION 2.1.2 (probabilistic program result checker) *A probabilistic program result checker for f is a probabilistic oracle program R_f which is used to verify,* for any program *P that supposedly evaluates f, that P outputs the correct answer on a given input in the following sense. On a given input x and confidence parameter α, R_f^P has the following properties:*

1. *If $P(x) \neq f(x)$ then R_f^P outputs "FAIL" (with probability $\geq 1 - \alpha$).*
2. *If P is a correct program for every input then R_f^P outputs "PASS" (with probability $\geq 1 - \alpha$).*

Some remarks about the model are in order:

- R_f^P is written specifically for the function f.

- The probabilities are with respect to a source of truly random independent bits available to the result checker, and *not* with respect to any assumptions about the input distribution. Thus, if $P(x) \neq f(x)$, the result checker catches that P is not correct with probability at least $1 - \alpha$ for any x.

- R_f is only allowed to access P as a black-box oracle. Therefore, this model by nature forces the checker to do something inherently different than program verification.

- If $P(x) = f(x)$ but P is faulty on other inputs then R_f is allowed to output either "FAIL" or "PASS", because the stronger requirement, that R_f always outputs "PASS" when $P(x) = f(x)$, is too strong in general. This is because these properties are supposed to hold for *any* program P that supposedly evaluates f. Consider the case when the range of f is $\{0, 1\}$, i.e. f is a decision function, and P is the trivial program that outputs 1 for all inputs. To satisfy the stronger requirement in this case would require R_f to be a correct program for f when making no calls to P, because P provides no information about f. Thus it would be impossible for R_f to be simpler than any correct program to evaluate f.

- Most result checkers satisfy a stronger condition than that in 1: If P has no bugs, then the result checker will always output "PASS". However, there are at least two known examples of result checkers where the weaker condition is needed. These are the matrix rank checker in [17, Blum Kannan],[39, Kannan], and the quadratic residuocity checker in [45, Kompella].

- We do not assume that the programs being checked are deterministic, we only require that their specifications require them to always be correct, i.e. often a probabilistic program may err with small probability and still be a correct program. See [39, Kannan] for a discussion of checking probabilistic programs.

- The model of computation typically used in this work will be a RAM bit cost model of computation as defined in [3, Aho Hopcroft Ullman].

A very important idea is to allow the result checker to call the program on other inputs while checking it on a given input. A priori, it would seem that the additional power to call the program does not help much, but due to recent advances in complexity theory and new notions of a mathematical proof ([15, Babai],[37, Goldwasser Micali Rackoff], [13, Ben-Or Goldwasser Kilian Wigderson], this often simplifies the checking process, and in some cases (for example the permanent problem) it is the only way that we know how to do polynomial time checking at all. Two reasons why allowing the result checker to call the program reduces the complexity of (sequential) checking are: Many decision problems are self-reducible, and a correct program can aid in solving the search problem, giving a proof of correctness. Secondly, it is noted in [15, Blum] [17, Blum Kannan], that the result checker can be thought of as a restricted version of an interactive proof as defined in [37, Goldwasser Micali Rackoff]. Intuitively, the program is proving to the user that the program has computed the result correctly.

2.1.1 Correctness of Result Checker

The problem remains of determining the correctness of the result checker. One of the primary reasons that the theory of checking was introduced is to make it possible to gain evidence that a program correctly computes a function f on given instances without trying to prove that the program correctly computes f on all inputs. However, the following question remains: "How do we know that a result checker for f is correct?" Blum does not propose to use program verification in the classical sense to prove that result checker is correct (although this may be possible for very simple result checkers, and of course for all the result checkers we develop we do give a proof of correctness that is supposed to convince the reader).

One possibility is to ask that the result checker be simpler than any correct program for computing f. This approach is attractive because intuitively if the result checker is simpler than any program for f it is also more likely to be correct, both in its specification, compilation into software, and in its hardware implementation. Moreover, since the same result checker can be used with respect to any program for f, it makes sense to spend a lot of energy writing a correct result checker, which can still be used even if a new (possibly faster) version of the program is implemented. This approach is similar to what is done when designing an operating system, where a tremendous amount of effort is spent building a simple and correct kernel.

The generally accepted notion of simplicity is the aesthetic simplicity of the result checker versus that of the fastest correct program. This notion of simplicity is often easily discernible by people but unfortunately is fairly unquantifiable. Although this is not quantifiable, intuitively the result checkers developed in this and other papers are much simpler in terms of the program code than the corresponding fastest program for the problem. For example, the integer multiplication checker presented here seems much simpler than multiplication programs based on Fast Fourier transforms, and the matrix multiplication checker seems much simpler than matrix multiplication programs based on the methods in [63, Strassen], [26, Coppersmith Winograd].

Blum suggests that we force the checker to be quantifiably different than any program for f. Then instead of relying on the correctness of the checker, we can rely on the hope that bugs in the checker are "independent" from bugs in the program and are unlikely to interact in such a way that bugs in the program will not be caught. We consider two quantifiable notions of programs begin different. These notions are based on limiting some resource of the result checker to enforce it to do something quantifiably different than the program. Thus far, most of the result checkers found which satisfy these quantifiable notions seem to be simpler than any program for the function as well. [15, Blum] introduces a notion of quantifiably different based on the running time requirements of the result checker versus that of the fastest correct program for f.

DEFINITION 2.1.3 (different) *We say that the result checker R_f is* quantifiably different *if, for all programs P, the incremental time of R_f^P is smaller than the running time of the fastest known program for computing f directly.* [1]

In the definition of *different*, we ignore the running time dependence on the confidence parameter β, which is typically a multiplicative factor of $O(\ln(1/\beta))$.[2]

Another quantifiable notion of difference that we consider is with respect to the algebraic model of computation (this notion is also considered in [66, Yao]). (Similar definitions can be made with respect to other structured models of computation.) Consider the problem of integer multiplication. Natural primitives to consider in writing a program for this this task are addition, the shift operation and comparisons. Intuitively, these primitives are simpler and easier to implement reliably individually than integer multiplication. Suppose we design a result checker for integer multiplication which only uses these primitives (and random bits), and makes calls to the purported program for integer multiplication. Of course, an integer multiplication program can also be designed using these same primitives. The *incremental operation count* of R_f on input x is the maximum over all P (including correct and faulty P) of the number of primitive operations used by R_f plus the number of calls to P. R_f is *quantifiably different in the algebraic sense* if the incremental operation count of R_f is asymptotically smaller than the smallest known number of primitive operations needed to implement integer multiplication. (We could also measure the algebraic complexity of R_f more carefully, keeping a count of each type of primitive operation instead of a lump sum.)

2.1.2 Efficiency

In the most straightforward applications of checking, whenever the program is executed the result checker is also executed. Thus, it is critical that the overhead cost of running the result checker doesn't neutralize the benefit from knowing that the output is correct (or the knowledge that the program is faulty).

DEFINITION 2.1.4 (efficient) *We say that result checker R_f is* efficient *if, for all programs P, the total time of R_f^P is linear in the running time of P and the input size.*

[1] Ideally, we would like the stronger requirement that the incremental time of R_f is asymptotically smaller than the running time of *any* correct program for f. However, for many interesting problems there is no known non-trivial lower bound on the running time of a correct program. Thus, as in [15, Blum], we adopt this more pragmatic definition of quantifiably different.

[2] 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 $\ln \alpha$.

DEFINITION 2.1.5 ($\alpha(n)$-efficient) *We say that result checker R_f is* $\alpha(n)$-efficient *if, for all programs P, the total time of R_f^P is at most $\alpha(n)$ times the running time of P and the input size.*

In the definition of *efficient*, we ignore the running time dependence on the confidence parameter β. We ask that the result checker be efficient, but failing that, we would like the result checker to be as efficient as possible.

2.2 Examples

Many result checkers have been found for various types of problems such as graph isomorphism, matrix rank, quadratic residuocity, linear programming, maximum matching, greatest common divisor, permanent and even PSPACE-complete problems [15, Blum], [23, Blum Raghavan], [17, Blum Kannan], [18, Blum Kannan Rubinfeld], [2, Adleman Huang Kompella] [50, Fortnow Karloff Lund Nisan] [62, Shamir]. We give two examples of result checkers: we show a result checker for integer sorting, and the result checker for matrix multiplication from [34, Freivalds].

2.2.1 Sorting

Consider the problem of sorting integers with the following specifications:

Input: A set of integers $X = \{x_1, x_2, ..., x_n\}$ (not necessarily distinct).

Output: The elements of X in sorted order: i.e. a list $y_1 \leq y_2 \leq ... \leq y_n$ such that $Y = \{y_1, ..., y_n\}$ is equal to X.

The best known algorithms for sorting require $O(n \log n)$ time. The result checker must verify that the output is in sorted order, and that the set of elements in the input list is the same as the set of elements in the output list. The first task is quite easy, but the second task is nontrivial, and on the algebraic decision tree model, is as difficult a task as sorting. In [15, Blum], [17, Blum Kannan] there are randomized algorithms for verifying that $X = Y$ which use hashing and run in $O(n)$ time. We present a deterministic algorithm which checks sorting in $O(n)$ time.

Checker Algorithm: (For simplicity, assume that n is a power of 2.)

```
$Y \leftarrow P(X)$
Do for $1 \leq i \leq n$
    append $\log n$ bits to the binary representation of the $i^{th}$ input
        indicating its location in the input list, i.e. $x'_i \leftarrow (x_i) \times n + i$.
        (Note that this does not affect the ordering of the elements.)
Let $X' = \{x'_1, \ldots, x'_n\}$.
```

```
$Y' \leftarrow P(X')$
Let $j$ be the last $\log n$ bits of $y'_i$, i.e. $j \leftarrow y'_i$ mod $n$.
Verify that $x'_j = y'_i$.
Verify that $x'_j$ has not been checked off yet and check off $x'_j$.
Let $Y'' = \{y'_1 \mathrm{div} n, \ldots, y'_n \mathrm{div} n\}$.
Verify that $Y$ is in sorted order, $|Y| = n$ and that $Y = Y''$.
If any verification fails, output "FAIL", else output "PASS".
```

2.2.2 Matrix Multiplication

Input: Integer n; $n \times n$ matrices A, B; β

Output: $C = A \cdot B$

The result checker presented here is due to Freivalds [34, Freivalds].

Specifications: On input 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 $O(n^2 \lceil \log(1/\beta) \rceil)$.

Checker Algorithm:

```
For $j = 1, \ldots, \lceil \log(1/\beta) \rceil$ do
  $r \leftarrow$ random $(n \times 1)$ 0/1 vector
  If $C \cdot r \neq A \cdot (B \cdot r)$ then output "FAIL" and RETURN
Output "PASS"
```

```
If $C \cdot R \neq A \cdot (B \cdot R)$ then output "FAIL" and RETURN
```

Proof: [of correctness] If $A \cdot B = C$, the result checker always outputs "PASS". Suppose $A \cdot B \neq C$. Let R be the set of $(n \times 1)$ 0/1 vectors. Let i, j be such that $(A \cdot B)_{ij} \neq C_{ij}$. Let $G = \{r | r \in R, A \cdot B \cdot r \neq C \cdot r\}$ and $\bar{G} = R - G$. We show a $1-1$ mapping from $\bar{G}$ to G, showing that $|G| \geq |\bar{G}|$, and thus $Pr_r[A \cdot B \cdot r \neq C \cdot r] \geq 1/2$. For $r = r_1 r_2 \ldots r_n \in \bar{G}$, $\sum_k (\sum_j a_{ij} b_{jl}) r_k = \sum_k c_{ik} r_k$ and $\sum_j a_{ij} b_{jl} \neq \sum_k c_{ik}$. Then for $\bar{r} = r_1 r_2 \ldots r_{l-1} \bar{r}_l r_{l+1} \ldots r_n \in G$. The mapping from r to $\bar{r}$ is $1-1$. After $\lceil \log(1/\beta) \rceil$ iterations, the result checker outputs "FAIL" with probability at least $1 - \beta$. ■

Chapter 3

Self-Testing/Correcting Programs

In this chapter we introduce the notion of *self-testing/correcting*, which is an extension of the theory of program checkers. The work in this chapter was done in collaboration with Mike Luby and Manuel Blum [20, Blum Luby Rubinfeld 2], [21, Blum Luby Rubinfeld 3], with the exception of the results in Section 3.5 which was done in collaboration with Madhu Sudan [56, Rubinfeld Sudan].

Consider any program P whose task is to evaluate a function f. A self-tester for f is a program that estimates the fraction of x for which $P(x) \neq f(x)$. We say that P is a "good heuristic" for f if this fraction is sufficiently large (a constant suffices for most purposes). For any input x, A self-correcting algorithm for f is a (probabilistic) algorithm that computes f correctly on every input (with high probability) when given access to any good heuristic for f. Thus, 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.

In slightly more detail, 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. As with result checkers, we insist that T_f be *different* than any correct program for computing f. This ensures that T_f must be doing something *quantifiably* different than computing f directly, because there is not enough time for this. A self-testing program is in this sense an "independent" verification step for a program P supposedly computing f. In addition, although it is hard to quantify, the self-testers we develop also have the property that the resulting code is aesthetically simple. We would like T_f to be *efficient*, in the sense that the running time of T_f, counting the time for calls to P, is within a constant multiplicative factor of the running time of P. This ensures that the advantages we gain by using T_f to self-test P are not overwhelmed by an inordinate running time slowdown.

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. As for self-testing programs and for the same reasons, we want C_f to be both *different* and *efficient.*

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 purportedly 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 ([26, Coppersmith Winograd], [63, Strassen], [60, Schonhage]). Thus, the self-testing/correcting pairs we develop for integer and matrix multiplication may be useful in practice.

We develop general techniques for constructing simple to program self-testing/correcting pairs for a variety of numerical functions. We show how our techniques apply to integer multiplication, the mod function, modular multiplication, integer division, polynomial multiplication, modular exponentiation, matrix multiplication and the evaluation of any fixed polynomial. Recently, [25, Cleve Luby] have shown how to apply these techniques to get self-testing/correcting pairs for the sine and cosine functions. It is not known how to solve any of these problems in linear time. The following table summarizes the running time behavior of our self-testing/correcting pairs as a function of the input size n. The second column is the incremental running time and the third column is the total running time, where $M(n)$ is the running time of P on inputs of size n. These times exclude a constant multiplicative factor and they also exclude the running time dependence on the confidence parameter β, which is typically $O(\ln(1/\beta))$.[1]

[1] In this thesis, $\ln \alpha$ denotes 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 $\ln \alpha$.

Problem	Incremental	Total
Integer Mult.	n	$M(n)$
Mod	n	$M(n)$
Mod Mult.	n	$M(n)$
Integer Div.	n	$M(n)$
Poly. Mult.	n	$M(n)$
Mod Exp., ϕ	n	$M(n)$
Mod Exp., no ϕ	$n \ln^4 n$	$M(n) \ln^3 n$
Matrix Mult.	n	$M(n)$
Deg. d Poly.	nd^5	$M(n)d^5$

Say that f is *close* to g if $f(x) = g(x)$ for most inputs. The theory of self-testing leads to interesting mathematical questions about properties that characterize functions which are close to a particular function. Suppose property Q characterizes the function f. We show that there is a property Q' which characterizes any function g which is close to f. 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 3.4 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. Section 3.5 shows similar results for functions which are close to functions that compute 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.

3.1 Related Work

[22, 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. [57, Rubinfeld] introduces checking for parallel programs, and uses self-testing to design a constant depth circuit to check the majority function (see section 6.3.1). We will see that a self-testing/correcting pair for a function f implies a program result checker for f. We will also see that a program result checker for f implies a self-tester for f, but it is not known whether a program result checker also implies a self-corrector. Previous to our work, [38, Kaminski] gives program result checkers for integer and polynomial multiplication. Independently of our work, [2, Adleman Huang Kompella] give program result checkers for integer multiplication and modular exponentiation. Both of these papers use very different techniques than ours. Previous to our work, [34, Freivalds] introduces a program result checker for matrix multiplication over a finite field.

[48, Lipton], independently of our work, discusses the concept of self-correcting programs and gives self-correctors for several functions. 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 [48, 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.

The techniques in this chapter have been applied to the theory of interactive proofs (see [37, Goldwasser Micali Rackoff], [15, Babai] and [13, Ben-Or Goldwasser Kilian Wigderson] for the discussion of interactive proofs). [53, Nisan] noted that the self-testing/correcting technique based on bootstrapping discussed in Section 3.6 can be combined with the observation about the permanent problem in [48, Lipton] (based on [9, Beaver Feigenbaum]) to construct a two-prover interactive proof system for the permanent problem. This led to the eventual discovery that *IP* = *PSPACE* ([50, Fortnow Karloff Lund Nisan], [62, Shamir], [7, Babai]).

The results in this chapter are related to those in [8, Babai Fortnow Lund]. In order to show that the multi-prover version of *IP* is equal to *NEXPTIME*, they 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. The incremental running time of their test, not counting the time for calls to P, runs in time polynomial in n and is independent of the number of terms in f. Combining the ideas in [8, Babai Fortnow Lund] with those in this chapter yields a self-tester for this same task which is much simpler, using only additions, comparisons and calls to P. A simple self-test is not a major issue with respect to the result in [8, Babai Fortnow Lund], where polynomial time is the major issue, but it is an important issue with respect to designing efficient self-testing/correcting pairs.

3.2 The Basics

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 $\mathcal{I}_n$. 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 $\mathcal{D}_n$. Let $\beta > 0$ be a confidence parameter.

Definition 3.2.1 (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 3.2.2 (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 β.*

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* $1 - \beta$.

$\epsilon_1 = 0$ is sufficient for assuring that either f will be computed correctly, or a fault in P will be detected. However, the value of ϵ_1 should be as close as possible to ϵ_2 to allow as faulty as possible heuristics P to pass test T_f^P and still have the self-corrector C_f^P work correctly.

Definition 3.2.3 (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 β. If* $\text{error}(f, P, \mathcal{D}_n) \leq \epsilon$ *then* $C_f^P(x) = f(x)$ *with probability at least* $1 - \beta$.

As with result checkers, we would like T_f and C_f to be both *different* and *efficient*, although sometimes 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 $O(\ln(1/\beta))$.

Definition 3.2.4 (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 $\mathcal{D}$.*

Because self-testers must be *different*, the algorithm 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 the value of $f(x)$ is not independently available, and P may be of no use in helping to compute it. 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.

One can generate random instances of f according to $\mathcal{D}$, and use the program result checker to verify that P is correct on those instances. Thus a program result checker for f also gives a self-tester for f. On the other hand, if one has a self-testing correcting pair for f, one can create a result checker for f on input x in the following manner: First use the self-tester to test P. If P fails, output "FAULTY". Otherwise, use the self-corrector to correctly compute $f(x)$ and compare the result with $P(x)$. If they agree, output "PASS", otherwise output "FAULTY".

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 3.2.5 (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 $\mathcal{D}_n$.*[2] *Informally, by* easily*, we mean that the worst case computation time of the random self-reduction (excluding the time for computing $f(a_1), \ldots, f(a_c)$) is smaller than that of computing f on inputs from $\mathcal{I}_n$.*

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 intervals of the integers. We often use the following notation.

DEFINITION 3.2.6 (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 3.2.7 (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 the same P as an oracle. 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 3.2.8 (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 $\mathcal{U}_{\{0\}}$ is the probability distribution such that zero has probability one. We let $x \in_{\mathcal{U}} X$ denote that x is randomly and uniformly distributed in X.*

[2] 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 $\mathcal{D}_n$.

3.3 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. We then give the generic self-correcting program that works for any random self-reducible function, and upon which all of the other self-correcting programs are based. For completeness, we then give the specific details of the self-correcting programs for integer multiplication, modular multiplication, modular exponentiation, integer division, matrix multiplication and polynomial multiplication. We also describe the result in [48, Lipton] which uses the same basic outline to develop a self-correcting program for any multivariate polynomial function over a finite field.

3.3.1 The Mod Function

We consider computing an integer $\bmod R$ for a positive number R. In this case, $f(x, R) = x \bmod R$. Assume that we have a program P such that 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 in $\{answer_m : m = 1, \ldots, N\}$
```

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 20), 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 running time, not counting calls to P, is linear in n, and it is also efficient, because the total running time, counting time for calls to P, is within a constant multiplicative factor of the running time of P.

3.3.2 Generic Self-Correcting Program

Let c be a positive integer and let f be any $c-$random self-reducible function (see page 20 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 in $\{answer_m : m = 1, \ldots, N\}$
```

Lemma 3 *The above program is a* $\frac{1}{4c}$*-self-correcting program for* f.

Proof: Because $\text{error}(f, P, \mathcal{D}_n) \leq \frac{1}{4c}$ and 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 lemma follows from Proposition 1. ■

3.3.3 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 in $\{answer_m : m = 1, \ldots, N\}$
```

Lemma 4 *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.

3.3.4 Modular Multiplication

We now consider multiplication of integers $\text{mod} R$ 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 $\text{error}(f, P, \mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\{R\}})$ $\leq 1/16$. 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 in $\{answer_m : m = 1, \ldots, N\}$
```

Lemma 5 *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. ■

3.3.5 Modular Exponentiation

We now consider exponentiation of integers mod$R$ 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 in $\{answer_m : m = 1, \ldots, N\}$
```

Lemma 6 *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 Chapter 5.

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 S' be the modular multiplication self-correcting program described in a previous subsection and let S 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 $\text{error}(f', P', \mathcal{U}_R \times \mathcal{U}_R \times \mathcal{U}_{\{R\}}) \leq 1/16$ then we can use S, making calls to P and making calls to S', which in turn makes to P', to self-correct f. Using this approach, the only operations computed by either S or S' are integer additions, comparisons and calls to the programs P and P'. The self-correcting program is different, because the running time, not counting calls to P or P', is linear in n, and it is also efficient, because the total running time, counting time for calls to 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. A third alternative is to use the library approach described above, but to use P to compute f' as follows: $P'(a, b, R) = (P(a + b, 2, R) - P(a, 2, R) - P(b, 2, R))/2$.

3.3.6 Integer Division

We now consider division of integers by R for a positive number R. In this case, $f(x, R) = (x \text{ div } R, x \bmod R)$. 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**$(R2^n, x, x_1, x_2, c)$
  $Divans_m \leftarrow (P_{div}(x_1, R) + P_{div}(x_2, R)) + (P_{mod}(x_1, R) + P_{mod}(x_2, R))$ 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 in $\{(Divans_m, Modans_m) : m = 1, \ldots, N\}$
```

Lemma 7 *The above program is a* $1/8$*-self-correcting program for integer division.*

Proof: Follows the outline of the proof of Lemma 3. ■

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 20) 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 $(P_{mod}(x_1, R) + P_{mod}(x_2, R))$ div R consists of one integer addition and one comparison.

3.3.7 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 error($f, P, \mathcal{U}_{M_{n \times n}[F]}$) $\leq 1/16$. 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 in $\{answer_m : m = 1, \ldots, N\}$
```

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

Proof: Follows the outline of the proof of Lemma 3. ■

3.3.8 Polynomial Multiplication

We consider multiplication of polynomials over a ring. Let $\mathcal{P}_{d[R]}$ denote the set of polynomials of degree d with coefficients from some ring R, and let $\mathcal{U}_{\mathcal{P}_{d[R]} \times \mathcal{P}_{d[R]}}$ be the uniform distribution on $\mathcal{P}_{d[R]} \times \mathcal{P}_{d[R]}$. In this case, $f(p(x), q(x)) = p(x) \cdot q(x)$, where $p, q \in \mathcal{P}_{d[R]}$.

Assume that we have a program P such that error$(f, P, \mathcal{U}_{\mathcal{P}_{d[R]} \times \mathcal{P}_{d[R]}}) \leq 1/16$. The following program is a $1/16$-self-correcting program for f making oracle calls to P with respect to $\mathcal{U}_{\mathcal{P}_{d[R]} \times \mathcal{P}_{d[R]}}$. The input to the program is $p, q \in \mathcal{P}_{d[R]}$ 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 $p_1 \in_{\mathcal{U}} \mathcal{P}_{d[R]}$
  Choose $q_1 \in_{\mathcal{U}} \mathcal{P}_{d[R]}$
  $p_2 \leftarrow p - p_1$
  $q_2 \leftarrow q - q_1$
  $answer_m \leftarrow P(p_1, q_1) + P(p_2, q_1) + P(p_1, q_2) + P(p_2, q_2)$
Output the most common answer in $\{answer_m : m = 1, \ldots, N\}$
```

Lemma 9 *The above program is a* $1/16$*-self-correcting program for polynomial multiplication.*

Proof: Follows the outline of the proof of Lemma 3. ■

3.3.9 Multivariate Polynomial Function

We consider the problem of computing any multivariate polynomial function over a finite field. In [48, Lipton], Lipton has shown a self-corrector for this problem based on the techniques of [9, Beaver Feigenbaum] which uses scalar multiplications and some preprocessing. We describe a similar self-corrector suggested by Mike Luby and Steven Omohundro that uses only additions and no preprocessing, and works for fields of prime cardinality. In this case, $f(x_1, \ldots, x_m, p) = q(x_1, \ldots, x_m) \bmod p$, for a prime p, multivariate polynomial q of degree d, and $x_1, \ldots, x_m$ in $\mathcal{Z}_p$.

In [48, Lipton], it is shown that in any finite field of size p, there are weights $\alpha_1, \ldots, \alpha_{d+1}$, such that for any polynomial $q(X)$ of degree $d < p$, and any x, t in the field,

$$\sum_{i=0}^{d+1} \alpha_i \cdot f(x + i \cdot t) = 0.$$

Lipton shows how to compute the α_i's by solving a system of linear equations. Since the α_i's are the same for *every* polynomial of degree d, this is something that can be done once in preprocessing. The self-corrector will need to perform multiplications by these α_i's. However, if p is prime, then using the method of differences described in [64, Van Der Waerden] (pg. 89), $\sum_{i=k}^{l} \alpha_i \cdot f(x + i \cdot t)$ can be computed using only $O((l-k)^2)$ additions at runtime. In this case, the α_i's turn out to be $\alpha_i = (-1)^i \begin{pmatrix} d+1 \\ i \end{pmatrix}$.

Assume that we have a program P such that error$(P, f, \mathcal{U}_{\mathcal{Z}_{p2^n}}{}^m \times \mathcal{U}_{\{p\}}) \leq \frac{1}{4(d+1)}$. The following program is a $\frac{1}{4(d+1)}$-self-correcting program for f making oracle calls to P with respect to $\mathcal{U}_{\mathcal{Z}_{p2^n}}{}^m \times \mathcal{U}_{\{p\}}$. The input to the program is n, p, $x_1, \ldots, x_m \in \mathcal{Z}_{p2^n}$ and the confidence parameter β.

Program Multivariate Polynomial Function Self-Correct $(n, p, x_1, \ldots, x_m, \beta)$

```
Do for $j = 1, \ldots, 12 \ln(1/\beta)$
  Randomly choose $t_1, \ldots, t_m$ independently according to $\mathcal{U}_{\mathcal{Z}_{p2^n}}$
   $answer_j \leftarrow \Sigma_{i=1}^{d+1} - \alpha_i \cdot P(x_1 +_{p2^n} i \cdot t_1, \ldots x_m +_{p2^n} i \cdot t_m) \bmod p$
Output the most common answer in $\{answer_j : j = 1, \ldots, 12 \ln(1/\beta)\}$
```

Lemma 10 *The above program is a $\frac{1}{4(d+1)}$-self-correcting program for any polynomial function of degree d.*

Proof: Follows the outline of the proof of Lemma 3. For $i \in \{1, \ldots, d+1\}$, $x_j +_{p2^n} i \cdot t_j$ is distributed according to $\mathcal{U}_{\mathcal{Z}_{p2^n}}$. ■

The incremental time is $O(d^2(n+p))$ and the total time is $O(d^2(n+p) + dT(n,p))$ where $T(n,p)$ is the running time of the program.

3.4 Linearity and Self-Testing

To highlight the importance of being able to self-test efficiently at runtime, consider the mod function. To self-correct on input x and modulus R, the assumption in [48, 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.

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.

3.4.1 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 satisfies the linearity property for most random tests is essentially a linear function, in the sense that there is some linear function which is equal to the original function on most of the domain.

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 1 *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 5 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.

3.4.2 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.

For simplicity, we assume that all groups are abelian; these results can be generalized to non-abelian groups as well [12, Ben-Or Coppersmith Luby Rubinfeld], but our applications are to abelian groups. Let G be a finite group with group operation $\circ$ and with c generators $g_1, \ldots, g_c$ and identity element 0. For $y \in G$, let y^{-1} denote the inverse of y. Let G' be a (finite or countable) group with group operation $\circ'$ and identity element $0'$. For $\alpha \in G'$, let α^{-1} denote the inverse of α. Let $f : G \rightarrow G'$ be a function. Intuitively, f is hard to compute compared to either $\circ$ or $\circ'$.

We say that f has the *linearity property* if:

(1) It is easy to choose $x \in_{\mathcal{U}} G$.

(2) F_{linear} is an easily computable function with the property that, 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.

(3) For each generator $g_i \in G$, F^i_{neighbor} is an easily computable function with the property that, 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.* This property is not needed for integer multiplication. 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 for all $z \in G$, $f(z \circ 1) = f(z) \circ' 1'$.

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' is an infinite group that has no finite subgroups except $\{0'\}$. 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 3.4.1, for modular multiplication and for modular exponentiation are all based on **Generic Self-Test 2**.

Program Generic Self-Test 1(ϵ, β)

```
$N \leftarrow \frac{72}{\epsilon} \ln(2/\beta)$
$t \leftarrow 0$
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)

```
    $N \leftarrow \frac{72}{\epsilon}\ln(4/\beta)$
    $t \leftarrow 0$
    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 motivation for why the self-testers work. For each $y \in G$, define the *discrepancy of* y to be

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

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 the linearity property, part (2), 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. If G' is infinite with no non-trivial finite subgroups then, because G is finite, h is the trivial mapping from G to $\{0'\}$.

Now suppose G' has a finite subgroup not equal to $\{0'\}$. Because of the linearity property, part (3), 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 $i = 1, \ldots, c$, $\text{disc}(z \circ g_i) = \text{disc}(z)$ 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:

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

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

Theorem 2 *Let G' be an infinite countable group with no finite subgroups except for the trivial subgroup $\{0'\}$. Then, $\delta \geq 2\psi/9$.*

Theorem 3 *Let G' be any (finite or countable) 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 2 and 3, due largely to Don Coppersmith, are simpler than our original proofs. A full exposition of some related general probability results will appear in [12, Ben-Or Coppersmith Luby Rubinfeld]. We now introduce some more notation and prove some intermediate lemmas that are used in the proofs of Theorems 2 and 3.

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 11, 12, 13 and 14, we assume that $\delta < 2/9$. Let δ' be defined as the smaller solution to the equality $\delta'(1-\delta') = \delta$. Because $\delta < 2/9$, $\delta' < 1/3$.

Lemma 11 *$\forall a \in G$, $\exists a' \in G'$ 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 $\delta'(1 - \delta') = \delta$. ■

Lemma 11 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 11.

Lemma 12 *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 11 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 13

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

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

Proof: By Lemma 12, 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 11 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 14 *Under the same conditions as (1) and (2) in Lemma 13,* $\Pr[\mathrm{disc}(x) = \mathrm{disc}(x \circ y)] \geq 1-\delta'$.

Proof: By Lemma 13, $h(a) = 0'$ for all $a \in G$. On the other hand, Lemma 11 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 2] Assume first that $\delta < 2/9$. By definition of δ and using Lemma 14, $\Pr[\mathrm{disc}(x) = \mathrm{disc}(x \circ y) = \mathrm{disc}(x) \circ' \mathrm{disc}(y)] \geq 1-\delta'-\delta$, 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 3] Analogous to the proof of Theorem 2. ■

Theorems 2 and 3 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 15 *Let G' be any (finite or countable) 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 16 *Let G' be any (finite or countable) 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 [55, Rènyi]. For a proof of this proposition, see for example [41, Karp Luby Madras].

Proposition 17 *Let $Y_1, Y_2, \ldots$ be independent identically distributed 0/1-valued random variables with mean μ. Let $\theta \leq 2$. If $N = \frac{1}{\mu} \cdot \frac{4\ln(2/\beta)}{\theta^2}$ 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 18 *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 4 **Generic Self-Test 1** *is an $(\epsilon/54, \epsilon)$-self-tester for any $0 \leq \epsilon \leq 1$.*

Proof:

($\psi \geq \epsilon$) By Theorem 2, 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 18, Part (1) yields $\Pr[\mathit{total}/N \leq \epsilon/9] \leq \beta$. On the other hand, if $\mathit{total}/N > \epsilon/9$ 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 15 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 18, Part (2), yields $\Pr[\mathit{total}/N \geq \epsilon/9] \leq \beta$. On the other hand, if $\mathit{total}/N < \epsilon/9$ 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 5 **Generic Self-Test 2** *is an $(\epsilon/54, \epsilon)$-self-tester for any $0 \leq \epsilon \leq 1$.*

Proof:

($\psi \geq \epsilon$) 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 $\delta_i \geq 1/2$. Case (1) is similar to the $\psi \geq \epsilon$ case of Theorem 4, using Theorem 3 in place of Theorem 2, which yields that the program outputs "FAIL" with probability at least $1 - \beta/2$. In case (2), because of the Corollary 18, Part (1), letting $\mu = 1/2$ and letting $N = 32\ln(4c/\beta)$ yields $\Pr[\mathit{total}'/N' \leq 1/4] \leq \frac{\beta}{2c}$. On the other hand, if $\mathit{total}'/N' > 1/4$ 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 $1-\beta$.

($\psi \leq \epsilon/54$) We partition the possibilities into two subcases: (1) For all $i = 1, \ldots, c$, $\delta_i \leq 1/8$; (2) There is an $i = 1, \ldots, c$ such that $\delta_i > 1/8$. A portion of case (1) is similar to the $\psi \leq \epsilon/54$ case of Theorem 4, which yields $\Pr[\mathit{total}/N > \epsilon/9] \leq \beta/2$. Also in case (1), using the Corollary 18, 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 $\Pr[\mathit{total}'/N' > 1/4] \leq \beta/2$. Thus, in case (1) the program outputs "PASS" with probability at least $1 - \beta$. In case (2), because of Lemma 16, there is some i such that $\delta_i > 1/8$ implies that $\psi > 1/16 > \epsilon/54$ since $\epsilon \leq 1$. Thus case (2) is impossible.

■

3.4.3 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 $x_1 + x_2 \geq 2^n$ 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 $F_{\text{linear}}(x_1, x_2) = yc2^n$ 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)

```
$N = 1152\ln(2/\beta)$
$\mathit{total} \leftarrow 0$
Do for $m = 1, \ldots, N$
  Call Int_Mult_Linear_Consistency$(n, \mathit{answer})$
  $\mathit{total} \leftarrow \mathit{total} + \mathit{answer}$
If $\mathit{total}/N > 1/144$ 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 6 *The above program is a* $(1/864, 1/16)$*-self-testing program for integer multiplication.*

Proof: Similar to the proof of Theorem 4, 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.

3.4.4 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, $F_{\text{linear}}(x_1, x_2, y) = 0'$.

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

Proof: See the proof of Theorem 5, and combine this with some of the aspects of the proof of Theorem 6. ■

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 discussed in Chapter 5 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', S') for the mod R function. The modular multiplication self-testing program can then call S' to compute $y \bmod R$. S' 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.

3.4.5 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, $F_{\mathrm{linear}}(x_1, x_2, y) = 0'$.)

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

Proof: Analogous to the proof of Theorem 7 (page 39). ■

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 20) 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 Section 3.6.3, we show how to reduce this requirement to the case where the mod function is not too faulty for $\bmod R$ computations if one assumes the existence of a program for modular inverse that is usually correct (since $a^{-1} \bmod R$ exists if and only if $\gcd(a, R) = 1$). Also in a later section of this chapter, 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.

3.4.6 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 $x_2 \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$
$x \leftarrow x_1 +_{R2^n} x_2$
$c \leftarrow (x_1 + x_2)$ 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 9 *The above program is a* $(1/864, 1/16)$*-self-testing program for integer division.*

Proof: Similar to the proof of Theorem 4 (page 36). ■

3.5 Self-Testing Polynomial Functions

We consider the problem of computing any polynomial function which maps to a finite field $\mathcal{Z}_p$ for prime p. In this case, $f(x,p) = q(x) \bmod p$, for a prime p, polynomial q of degree d, and x in $\mathcal{Z}_{p2^n}$. We have already seen a self-corrector for the more general problem of computing any multivariate polynomial function. We show a $(\frac{\epsilon^2}{4(d+2)}, \epsilon)$-self-testing program ($\epsilon \leq \frac{1}{10(d+1)^2}$) for f with respect to $\mathcal{U}_{\mathcal{Z}_{p2^n}} \times \mathcal{U}_{\{p\}}$. The input to the program is n, p, a list of $d+1$ inputs $a_1, \ldots, a_{d+1}$, the corresponding values of the function at these inputs $b_1, \ldots, b_{d+1}$, and the confidence parameter β.

Part of the tester is a subroutine, called **Degree_Test**, that tests whether there is a polynomial g of degree d such that $\Pr_{x \in \mathcal{Z}_{p2^n}}[P(x) = g(x)] \geq 1 - \epsilon$. Once this is verified, the other part of the tester indirectly verifies, by making calls to P, that the polynomial g is actually equal to f, by verifying that they are equal on at least $d+1$ inputs.

Let $\alpha_0, ..., \alpha_d$ be as discussed in the section on self-correcting any multivariate polynomial function. Recall that $\alpha_0 = 1$ and that $\sum_{i=0}^{d+1} \alpha_i = 0$.

Program Polynomial Self-Test $(n, d, p, (a_1, b_1), \ldots, (a_{d+1}, b_{d+1}), \beta)$

```
  $\beta' \leftarrow \beta/(2d+2)$
  Call Degree_Test($n, d, p, \beta/2$)
  For $k = 1, \ldots, d+1$, call Equality_Test($n, d, p, (a_k, b_k), \beta'$)
  Output "PASS"
```

Degree_Test (n, d, p, β)

```
  $total \leftarrow 0$
  $N \leftarrow \frac{16}{\epsilon^2} \ln(2/\beta)$
  Do for $j = 1, \ldots, N$
    Randomly choose $x, t \in \mathcal{U}_{\mathcal{Z}_{p2^n}}$
    If $\Sigma_{i=0}^{d+1} \alpha_i \cdot P(x +_{p2^n} it) \bmod p \neq 0$ then $total \leftarrow total + 1$
  If $total/N > \epsilon^2/2$ then output "FAIL" and halt.
```

Equality_Test $(n, d, p, (a, b), \beta)$

```
  $total \leftarrow 0$
  $N \leftarrow 12 \ln(1/\beta)$
  Do for $j = 1, \ldots, N$
    Randomly choose $t \in \mathcal{U}_{\mathcal{Z}_{p2^n}}$
    If $b \neq \Sigma_{i=1}^{d+1} - \alpha_i \cdot P(a +_{p2^n} it) \bmod p$
      then $total \leftarrow total + 1$
  If $total/N > 1/4$ then output "FAIL" and halt.
```

Theorem 10 *The above program is a* $(\frac{\epsilon^2}{8(d+2)}, \epsilon)$*-self-testing program for the polynomial function for* $\epsilon \leq \frac{1}{10(d+1)^2}$.

Let $\delta = \Pr_{x,t \in \mathcal{Z}_{p2^n}}[\Sigma_{i=0}^{d+1} \alpha_i P(x +_{p2^n} it) \bmod p \neq 0]$. Let δ' be the solution to $\delta'^2 = \delta$. Let $\psi = \Pr_{x \in \mathcal{Z}_{p2^n}}[P(x) \neq f(x)]$.

In order to prove Theorem 10, we first prove the following:

Theorem 11 *If* $\delta \leq \frac{1}{100(d+1)^4}$ *then there is some polynomial* g *of degree* d *such that* $\Pr_x[P(x) = g(x)] \geq 1 - \delta'$.

Proof: [of Theorem 11] Define the set of good values as : $\mathrm{G} = \{x | \Pr_t[\sum_{i=0}^{d+1} \alpha_i P(x +_{p2^n} it) \bmod p = 0] \geq 1 - \delta'\}$.

The following lemma is proved by a simple counting argument:

Lemma 19 $|\mathrm{G}| \geq 1 - \delta'$.

Lemma 20 *Let $\delta'' \leftarrow 4(d+1)\delta'$. Then for all $x \in \mathcal{Z}_{p2^n}$ there exists an $x' \in \mathcal{Z}_p$ such that $\Pr_{t \in \mathcal{Z}_{p2^n}}[x' = \sum_{i=1}^{d+1} -\alpha_i P(x +_{p2^n} it) \bmod p] \geq 1 - \delta''$.*

Proof: [of Lemma 20] We note that for all $x \in \mathcal{Z}_{p2^n}$:

$$\begin{aligned}
\Pr_{t_1, t_2 \in \mathcal{Z}_{p2^n}}[\sum_{i=1}^{d+1} \alpha_i P(x +_{p2^n} it_1) \bmod p &= \sum_{i=1}^{d+1} \alpha_i \sum_{j=1}^{d+1} -\alpha_j P(x +_{p2^n} it_1 +_{p2^n} jt_2) \bmod p && (1) \\
&= \sum_{j=1}^{d+1} \alpha_j \sum_{i=1}^{d+1} -\alpha_i P(x +_{p2^n} it_1 +_{p2^n} jt_2) \bmod p && (2) \\
&= \sum_{j=1}^{d+1} \alpha_j P(x +_{p2^n} jt_2) \bmod p] \geq 1 - 4(d+1)\delta' && (3)
\end{aligned}$$

Using the definitions of G and Lemma 19, equation (1) holds with probability $\geq 1 - 2(d+1)\delta'$ since for any $i \in [1, \ldots, d+1]$, $\Pr_{t_1 \in \mathcal{Z}_{p2^n}}[(x +_{p2^n} it_1) \in \mathrm{G}$ and $P(x +_{p2^n} it_1) = \sum_{j=1}^{d+1} -\alpha_i P((x +_{p2^n} it_1) +_{p2^n} jt_2) \bmod p] \geq 1 - 2\delta'$. Similarly, equation (3) holds with probability $\geq 1 - 2(d+1)\delta'$. Since the probability that the same object is drawn twice in two independent trials lower bounds the probability of drawing the most likely object, the lemma follows. ■

Lemma 19 leads to the definition of the function g from $\mathcal{Z}_{p2^n}$ to $\mathcal{Z}_p$ defined as follows: For all $x \in \mathcal{Z}_{p2^n}$, let $g(x) \leftarrow x'$, where x' is the element of $\mathcal{Z}_p$ described in Lemma 20.

Comment: Note that for all $x \in G$, $x' = P(x)$ and so $Pr_{x \in \mathcal{Z}_{p2^n}}[P(x) = g(x)] \geq 1 - \delta'$.

Lemma 21 *For all $x \in \mathcal{Z}_{p2^n}$, $\Pr_{t \in \mathcal{Z}_{p2^n}}[\sum_{i=0}^{d+1} \alpha_i g(x +_{p2^n} it) \bmod p = 0] \geq 1 - \delta'' - (d+1)\delta'$.*

Proof: [of Lemma 21] By Lemma 20, $\Pr_{t \in \mathcal{Z}_{p2^n}}[g(x) = \sum_{i=1}^{d+1} -\alpha_i P(x +_{p2^n} it) \bmod p] \geq 1 - \delta''$. By the comment following Lemma 20, $\Pr_{t \in \mathcal{Z}_{p2^n}}[x +_{p2^n} jt \in G] \geq 1 - \delta'$, and thus $\Pr_{t \in \mathcal{Z}_{p2^n}}[\forall j = 1, \ldots, d+1, P(x +_{p2^n} jt) = g(x +_{p2^n} jt)] \geq 1 - (d+1)\delta'$. ■

Lemma 22 *g is a polynomial of degree $\leq d$.*

Proof: [of Lemma 22] It is sufficient to show that $\forall x, t \ \sum_{i=0}^{d+1} \alpha_i g(x +_{p2^n} it) \bmod p = 0$ [64, Van Der Waerden] p.89. We first upper bound for each $i = 0, \ldots, d+1$ the quantity $\Pr_{t' \in \mathcal{Z}_{p2^n}}[g(x +_{p2^n} it) = \sum_{j=1}^{d+1} -\alpha_i g(x +_{p2^n} it +_{p2^n} ijt') \bmod p]$. For $i = 0$, this occurs with probability 1. For $i \neq 0$, ijt' is random, and thus by Lemma 21, this probability is at least $1 - \delta'' - (d+1)\delta'$. Thus

$$\begin{aligned}
\Pr_{t' \in \mathcal{Z}_{p2^n}}[\sum_{i=0}^{d+1} \alpha_i g(x +_{p2^n} it) \bmod p &= \sum_{i=0}^{d+1} \alpha_i \sum_{j=1}^{d+1} -\alpha_j g(x +_{p2^n} it +_{p2^n} ijt') \bmod p \\
&= \sum_{j=1}^{d+1} -\alpha_j \sum_{i=0}^{d+1} \alpha_i g(x +_{p2^n} i(t +_{p2^n} jt')) \bmod p] \\
&\geq 1 - (d+1)(\delta'' + (d+1)\delta') > 0 \quad (1)
\end{aligned}$$

Finally, using Lemma 21 again, and the fact that jt' is random for $j \neq 0$, for all $j = 1, \ldots, d+1$, $\Pr_{t' \in \mathcal{Z}_{p2^n}}[\sum_{i=0}^{d+1} \alpha_i g(x +_{p2^n} i(t +_{p2^n} jt')) \bmod p = 0] \geq 1 - \delta'' - (d+1)\delta'$. Thus, summing over all j and combining this with (1), $\Pr_{t' \in \mathcal{Z}_{p2^n}}[\sum_{i=0}^{d+1} \alpha_i g(x +_{p2^n} it) \bmod p = 0] \geq 1 - 2(d+1)(\delta'' + (d+1)\delta') \geq 0$. Since the probability is positive and t' does not appear in the expression, $\sum_{i=0}^{d+1} \alpha_i g(x +_{p2^n} it) \bmod p = 0$. ■

(end of proof of Theorem 11) ■

Proof: [of Theorem 10]

($\psi \geq \epsilon$) (should output "FAIL") Let g be defined as the degree d polynomials for which the quantity $\Pr_x[P(x) \neq g(x)]$ is minimized. Let $\psi' = \Pr_x[P(x) \neq g(x)]$.

($\psi' \geq \epsilon$) By the assumption and Theorem 11, $\delta \geq \epsilon^2$. Then letting $\mu = \epsilon^2$ and $N = \frac{16}{\mu} \ln(4/\beta)$, and using the Corollary 18, Part (1) we see that in the degree test, $\Pr[total/N \leq \epsilon^2/2] \leq \beta/2$. On the other hand, if $total/N > \epsilon^2/2$ then the output of the degree test is "FAIL". Thus, if $\psi \geq \epsilon$ and $\psi' \geq epsilon$, the degree test outputs "FAIL" with probability at least $1 - \beta/2$.

($\psi' \leq \epsilon$) Since $\psi \neq \psi'$, $g \neq f$. Then there is an $i \in [1..d+1]$ such that $f(a_i) \neq g(a_i)$. From the proof of Theorem 11 we see that if a polynomial g of degree d such that $\Pr_x[P(x) = g(x)] \geq 1 - \epsilon$ exists, we can compute $g(x)$ by choosing a random $t \in \mathcal{Z}_{p2^n}$ and computing $g(x)$ as $\sum_{i=1}^{d+1} -\alpha_i P(x +_{p2^n} it) \bmod p$ with probability of error at most $(d+1)\delta' \leq 1/4$ in each pass. Thus the equality test will fail with probability $\geq 1 - \beta'$.

($\psi \leq \epsilon^2/4(d+2)$) Using a proof similar to the second part of Theorem 4 (p. 36), one can show that if $\Pr_x[P(x) = g(x)] \geq 1 - \frac{\epsilon^2}{4(d+2)}$, then the degree test passes with probability $\geq 1 - \beta/2$, and each equality test passes with probability $\geq 1 - \beta'$.

The total error probability of the degree test and all of the equality tests is $\leq \beta/2 + (d+1)\beta' = \beta$.

■

3.6 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 functions 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 3.6.1 (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 $\mathcal{I}_{n-1}$. 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 22. 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:
    Randomly generate $a_1, \ldots, a_c$ from $x$
    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 23 **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, $1 \leq i \leq l$, 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 12 **Generic Bootstrap Self-Test** *meets the specifications.*

Proof:

(1) If there is an i, $1 \leq i \leq l$ such that for all $1 \leq j \leq i - 1$, 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 $1 - \beta/l \geq 1 - \beta$.

(2) If, for all i, $1 \leq i \leq l$, 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 $1 - \beta$.

■

3.6.1 Matrix Multiplication

We showed in Subsection 2.2.2 how to get a self-tester for matrix multiplication. To illustrate the method, we show in this subsection how to get another self-tester based on bootstrapping.

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 $M_{n\times n}[F]$.

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

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

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.

3.6.2 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.

3.6.3 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 in Chapter 5, 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. A modular multiplication can also be easily computed using a program that correctly computes modular exponentiation (and in particular is able to square) using the fact that $x \cdot y \bmod R = ((x+y)^2 - x^2 - y^2)/2 \bmod R$. Thus, the ideas of the library approach can be applied to this problem, without assuming the existence of any programs for other problems.

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 $R > 3$, $\Phi(R) = |\mathcal{Z}_R^*| > \frac{R}{6\ln(n)}$ [56, 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 24 *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.

3.6.4 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, but with the assumption that we have access to P', a program for modular inverse, where $P'(w, R) \cdot_R w = 1$ with probability at least $\frac{1}{c \ln(n)}$ when $w \in_{\mathcal{U}} \mathcal{Z}_R$. 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 25 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 26 **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 27 **Mod_Expon Bootstrap Self-Test** *meets the specifications.*

Proof: Similar to the proof of Theorem 12 (page 48). ■

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.

[2, 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.

Chapter 4

Approximate Result Checking and Self-Testing/Correcting

In the notions of a result checker and self-testing/correcting pair considered so far, a result of a program is considered incorrect if it is not *exactly* equal to the function value. Some programs are designed only to correctly *approximate* the value of a function. In this chapter we introduce approximate result checkers and approximate self-testers/correctors. An approximate result checker checks that the program correctly approximates the function on a particular input. Similarly, an approximate self-tester checks that the program correctly approximates the function on most inputs, and an approximate self-corrector takes a program that approximates the function on most inputs, and turns it into a program that approximates the function on all inputs. All of the functions discussed in this chapter map the domain into $\mathcal{Z}$ or $\mathcal{Z}_q$ for some positive integer q. The notions can be naturally extended for functions whose ranges are metric spaces.

These results also apply to functions which are approximations of other functions, e.g. the quotient function $f'(x, R) = x \text{ div } R$ can be thought of as an approximation of the integer division function $f(x, R) = (x \text{ div } R, x \bmod R)$. We have already seen how to apply self-testing/correcting to integer division, but we do not know how to devise a self-testing/correcting program for the quotient function. On the other hand, in this and other applications, instead of the exact answer, a reasonably good approximation of the exact answer is all that is required. This function is important because it is a system function that is often used.

We present a generic technique for designing approximate self-correctors for all of the functions with the random self-reducibility property, as well as the quotient function. We present a generic technique for designing approximate result checkers/self-testers for all of the functions with the linearity property which map to $\mathcal{Z}$, as well as the quotient function. We know that a self-testing/correcting pair can be easily made into a result checker, and similarly an approximate self-testing/correcting pair can be easily made into an approximate result-checker. Thus we concentrate on the approximate self-testing/correcting pairs.

We assume that there is a well-defined metric space as the range of f in the following definitions.

DEFINITION 4.0.2 *We use* $a \approx_\zeta b$ *to denote that* $|a - b| \leq \zeta$. *Then* $a \approx_{\zeta_1} b \approx_{\zeta_2} c$ *implies that* $a \approx_{\zeta_1+\zeta_2} c$.

DEFINITION 4.0.3 *We say that* x *is* $\Delta - good$ *if* $P(x) \approx_\Delta f(x)$, *otherwise, we say that* x *is* $\Delta - bad$.

DEFINITION 4.0.4 (approximate result checker) *Let* $0 \leq \Delta_1 \leq \Delta_2$. *A* (Δ_1, Δ_2)-approximate result checker *for* f *is a probabilistic oracle program* R_f *that has the following properties for any program* P *on input* x *and* β.

1. *If* x *is* $\Delta_2 - bad$ *then* R_f *outputs "FAULTY" with probability* $\geq 1 - \beta$.
2. *If* P *is* $\Delta_1 - good$ *on all inputs, then* R_f *outputs "OK" with probability* $\geq 1 - \beta$.

DEFINITION 4.0.5 (approximate error) *Let the* Δ-approximate error *of* g *and* f *with respect to* $\mathcal{D}$ *be defined as* $\text{apperr}(f, g, \mathcal{D}, \Delta) = \Pr_{x \in \mathcal{D}}[x \textit{ is } \Delta - bad]$.

DEFINITION 4.0.6 (approximates) *If* $\text{apperr}(f, g, \mathcal{D}, \Delta) \leq \epsilon$ *then we say that* g (ϵ, Δ)-approximates $(f, \mathcal{D})$.

DEFINITION 4.0.7 (approximate self-testing) *Let* $0 \leq \epsilon_1 < \epsilon_2 \leq 1$. *Let* $0 \leq \Delta_1 \leq \Delta_2$. *An* $(\epsilon_1, \epsilon_2, \Delta_1, \Delta_2)$-approximate 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* β.

1. *If* P (ϵ_1, Δ_1)*-approximates* $(f, \mathcal{D})$ *then* T_f^P *outputs "PASS" with probability at least* $1 - \beta$.
2. *If* P *does not* (ϵ_2, Δ_2)*-approximate* $(f, \mathcal{D})$ *then* T_f^P *outputs "FAIL" with probability at least* $1 - \beta$.

The value of ϵ_1 should be as close as possible to ϵ_2 and the value of Δ_1 should be as close as possible to Δ_2 to allow as faulty as possible programs P to pass test T_f^P and still have the approximate self-corrector C_f^P work correctly.

Definition 4.0.8 (approximate self-correcting) *Let* $0 \leq \epsilon < 1$, $\Delta_1 \leq \Delta_2$. *An* $(\epsilon, \Delta_1, \Delta_2)$-approximate 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* β. *If* P (ϵ_1, Δ_1)-*approximates* $(f, \mathcal{D})$ *then for all* x, $C_f^P(x) \approx_{\Delta_2} f(x)$ *with probability at least* $1 - \beta$.

We would like T_f and C_f to be both *different* and *efficient* as discussed previously.

Definition 4.0.9 (approximate self-testing/correcting pair) *Let* $\Delta_1 \leq \Delta_2 \leq \Delta_3$. *A* $(\Delta_1, \Delta_2, \Delta_3)$-approximate self-testing/correcting pair *for* f *is a pair of probabilistic programs* (T_f, C_f) *such that there are constants* $0 \leq \epsilon_1 < \epsilon_2 < 1$ *and an ensemble of distributions* $\mathcal{D}$ *such that* T_f *is an* $(\epsilon_1, \epsilon_2, \Delta_1, \Delta_2)$-*approximate self-testing program for* f *with respect to* $\mathcal{D}$ *and* C_f *is an* $(\epsilon_2, \Delta_2, \Delta_3)$-*approximate self-correcting program for* f *with respect to* $\mathcal{D}$.

It is easy to see that a $(\Delta_1, \Delta_2, \Delta_3)$-*approximate self-testing/correcting pair* for f can be turned into a $(\Delta_1, 2\Delta_3)$-*approximate result checker*

We often use the following corollary to Proposition 1 (page 21) in the proofs of the lemmas in this section.

Corollary 28 *Let* a *and* b *be real numbers such that* $a \leq b$. *Let* $y_1, \ldots, y_m$ *be independent real-valued random variables such that for each* $i = 1, \ldots, m$, $\Pr[a \leq y_i \leq b] \geq 3/4$. *Let median be the index of the median value of* $y_1, \ldots, y_m$. *Then*

$$\Pr\left[a \leq y_{median} \leq b\right] \geq 1 - e^{-m/12}.$$

Proof: For each $i = 1, \ldots, m$, define $x_i = 1$ if $a \leq y_i \leq b$ and $x_i = 0$ otherwise. Suppose y_{median} is not between a and b; without loss of generality suppose the $y_{median} < a$. Then for at least $m/2$ of the i, $y_i < a$ and consequently for at least $m/2$ of the i, $x_i = 0$. But by Proposition 1, this happens with probability at most $e^{-m/12}$. ■

4.1 Approximate Self-Correcting

In a previous chapter, the property of random self-reducibility was used in designing self-testers/correctors. By knowing the value of the function on random inputs, this property allows one to easily compute the function on a particular input. Consider the situation in which the approximate value of the function is known on random inputs. Then, can the function be approximated on a particular input? If this is true, we say that the function is *approximately random self-reducible*. We formalize the definition of approximate random self-reducibility.

Definition 4.1.1 (approximate random self-reducibility) *Let $x \in \mathcal{I}_n$. Let $c > 1$ be an integer. We say that f is (Δ_1, Δ_2)-approximate $c-$random self-reducibility if there is an easily computable function F_{appran} of x, $a_1, \ldots, a_c$ and $g(a_1), \ldots, g(a_c)$, where $a_1, \ldots, a_c$ are easily computable given x, each a_i is randomly distributed in $\mathcal{I}_n$ according to $\mathcal{D}_n$.*[1] *F_{appran} has the property that if, for all $i = 1, \ldots, c, g(a_i) \approx_{\Delta_1} f(a_i)$ then $f(x) \approx_{\Delta_2} F_{\mathrm{appran}}(x,\ a_1,\ \ldots,\ a_c,\ g(a_1),\ \ldots,\ g(a_c))$. By* easily, *we mean that the worst case computation time of the approximate random self-reduction (excluding the time for computing g on $a_1, \ldots, a_c$) is smaller than that of computing $f(x)$ on inputs from $\mathcal{I}_n$.*

The strength of this property is that it can be used to transform a program that approximates a function on a large enough fraction of the inputs into a program that approximates $f(x)$ with high probability for *every* input x.

Integer division $f(x, R) = (x \text{ div } R, x \bmod R)$ was shown to be random self-reducible as follows: Given $x \in \mathcal{Z}_{R2^n}$, choose $x_1 \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$ and let $x_2 \in \mathcal{Z}_{R2^n}$ be such that $x = x_1 +_{R2^n} x_2$. Then $x \bmod R = x_1 \bmod R +_R x_2 \bmod R$, and $x \text{ div } R = x_1 \text{ div } R + x_2 \text{ div } R - ((x_1 + x_2) \text{ div } 2^n R)2^n + (x_1 \bmod R + x_2 \bmod R) \text{ div } R$ (recall that the div by $2^n R$ and the last div R are easily computable). However, the quotient function $f(x, R) = x \text{ div } R$ is not quite random self-reducible by this relationship because the quotient function does not provide $x_1 \bmod R$ and $x_2 \bmod R$. On the other hand, we have that $x \text{ div } R = x_1 \text{ div } R + x_2 \text{ div } R - ((x_1 + x_2) \text{ div } 2^n R)2^n + \zeta$ where $\zeta \in \{0, 1\}$. There is no obvious way to easily compute ζ using only the quotient function, though using this relationship, we can still *approximate* $x \text{ div } R$ as $x_1 \text{ div } R + x_2 \text{ div } R - ((x_1 + x_2) \text{ div } 2^n R)2^n$ and know that our approximation is within 1 of the correct answer. Furthermore, if we have approximations to $x_1 \text{ div } R$ and $x_2 \text{ div } R$ then we can use this relationship to approximate $x \text{ div } R$. From this it is easy to see that the quotient function is $(\Delta, 2\Delta + 1)$-approximately random self-reducible. Thus we have an example of a function which is not known to be random self-reducible, but is known to be approximately random self-reducible.

In the following subsections, we show the specific details of the approximate self-correcting program for the quotient function. We then give the generic approximate self-correcting program that works for any approximate random self-reducible function, and upon which the all the mentioned approximate self-correcting programs are based.

4.1.1 Quotient

We consider computing the quotient of an integer when divided by a positive number R. In this case, $f(x, R) = x \text{ div } R$. In a subsequent section, we give an approximate self-testing algorithm for the quotient function. Suppose that the $x \in \mathcal{Z}_{R2^n}$. Assume that we have a program P such that P $(1/8, \Delta)$-approximates $(f, \mathcal{U}_{\mathcal{Z}_{R2^n}} \times \mathcal{U}_{\{R\}})$. We have seen that

[1] 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 $\mathcal{D}_n$.

the quotient function is $(\Delta, 2\Delta+1)$-approximately random self-reducible. The following program is a $(1/8, \Delta, 2\Delta+1)$-approximate 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 Quotient Function Approximate Self-Correct(n, R, x, β)

```
$N \leftarrow 12 \ln(1/\beta)$
Do for $m = 1, \ldots, N$
   Call **Random_Split**$(R2^n, x, x_1, x_2, c)$
   $answer_m \leftarrow P(x_1, R) + P(x_2, R) - c \cdot 2^n$
Output median of $answer_1, \ldots, answer_N$
```

Lemma 29 *The above program is a* $(1/8, \Delta, 2\Delta+1)$*-approximate self-correcting program for the quotient function.*

Proof: Follows the outline of the proof of Lemma 30 (page 60). For $i \in \{1,2\}$, $x_i \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$. Thus, by the properties of P, x_i is Δ-bad with probability at most $1/8$, and consequently both calls to P in a single loop return a Δ-good answer with probability at least $3/4$. Since the quotient function is $(\Delta, 2\Delta+1)$-approximate random self-reducible, if both x_1 and x_2 are Δ-good then $f(x) \approx_{2\Delta+1} answer_m$. The lemma follows by a straightforward application of Corollary 28. ■

The quotient function approximate self-correcting program is very simple to code, the only operations used are integer additions, comparisons and calls to the program P. Note that the approximate self-correcting program is different, because the running time, not counting calls to P, is linear in n, and it is also efficient, because the total running time, counting time for calls to P, is within a constant multiplicative factor of the running time of P.

4.1.2 Generic Approximate Self-Correcting Program

Generic Self-Correct is a $(\frac{1}{4c}, \Delta_1, \Delta_2)$-approximate self-correcting program for any (Δ_1, Δ_2)-approximate random self-reducible function f with respect to $\mathcal{D}_n$. Let R be such that f maps to $\mathcal{Z}_R$ (if f maps to $\mathcal{Z}$, we make the convention that $R = \infty$). We assume $\Delta_2 < R/8$.

Program Generic Approximate Self-Correct(n, R, 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 **Find-Modular-Median**$(N, answer_1, \ldots, answer_N, R)$
```

Function Find-Modular-Median$(l, a_1, a_2, \ldots, a_l, q)$

```
    If $q = \infty$, output median of $a_1, \ldots, a_l$ and return.
    If majority of $i$ satisfy $(0 \leq a_i \leq q/4$ or $3q/4 \leq a_i \leq q)$
        then $split \leftarrow q/2$
        else $split \leftarrow 0$
    Do for $i = 1, \ldots, l$
        $b_i \leftarrow a_i +_q split$
    $c \leftarrow$ median of $b_1, \ldots, b_l$
    Output $c -_q split$
```

Lemma 30 **Generic Self-Correct** *is a* $(\frac{1}{4c}, \Delta_1, \Delta_2)$*-approximate self-correcting program for* f *with respect to* $\mathcal{D}_n$*.*

Proof: Because P $(\frac{1}{4c}, \Delta_1)$-approximates $(f, \mathcal{D}_n)$, and because for each $k = 1, \ldots, c$, a_k is randomly distributed in $\mathcal{I}_n$ according to $\mathcal{D}_n$, all c of the a_i's are Δ_1-good with probability at least $3/4$ each time through the loop. If all c of the a_i's are Δ_1-good, then by the (Δ_1, Δ_2)-approximate random self-reducibility property of f with respect to $\mathcal{D}_n$, $f(x) \approx_{\Delta_2} answer_m$. If $R = \infty$, then the median is within Δ_2 of $f(x)$ by a straightforward application of Corollary 28. If R is finite, since modular numbers can be thought of as lying on a circle, there is no notion of a median. The idea is to choose a split point along the circle that is not within Δ_2 of $f(x)$. We then treat the points as lying on a total order, where the minimum is the first point after the split point and the maximum is the last point before the split point. We find the median with respect to this order. By Proposition 1 (page 21), the majority of the answers lie within Δ_2 of $f(x)$ with probability at least $1 - \beta$. Assume the majority of the answers are within Δ_2 of $f(x)$. If the circle is split at a point somewhere further than Δ_2 from $f(x)$, using reasoning similar to that in the proof of Corollary 28, the median of the answers with respect to the order defined by the split will lie within Δ_2 of $f(x)$. If $f(x) \approx_{\Delta_2} 0$ then because $\Delta_2 < R/8$, $split \leftarrow R/2$ and thus the median of the answers with respect to the order defined by the split is within Δ_2 of $f(x)$. Similarly, if $f(x) \approx_{\Delta_2} R/2$, $split \leftarrow 0$ and thus the median of the answers is within Δ_2 of $f(x)$. If $f(x)$ is not within Δ_2 of either 0 or $R/2$, then regardless of where the split is made, the median of the answers is within Δ_2 of $f(x)$. ■

4.2 Approximate Linearity and Approximate Self-Testing

We saw that functions which have the linearity property can be self-tested. In this section we show that programs which approximate any linear function which maps to the integers can be approximate self-tested. As mentioned before in the discussion of approximate self-correcting, there are functions such as the quotient function, where the linearity property does not hold, but something very close to it does. We call such functions *approximately linear*. We show that programs which approximate a linear mapping from a given domain to the integers can be also be approximate self-tested. This method applies to all linear functions which map to the integers as well as the quotient function. All of the results in this section are easily stated for linear or approximately linear functions that map to infinite cyclic groups.

To give some idea of how the method works, we concentrate on the quotient function. We define the approximate linearity property for functions which map to the integers, and give a generic approximate tester that works for any function with this property (and thus works for any function with the linearity property as well).

4.2.1 Quotient Function

For positive integers x and R, let $f_R(x) = x$ div R. (As before, we are viewing this as a function of one input x, where R is a fixed but arbitrary positive integer.) Because the approximate self-correcting program for the quotient function relies on a program that is approximately correct for most inputs with respect to a particular R, the approximate self-testing program for the quotient function is designed to approximate self-test with respect to an input divisor R. This is an important motivation for constructing efficient approximate self-testing programs, because the approximate self-testing program is executed each time a new divisor is used.

There is one critical test performed by the approximate self-tester. Let x_1 and x_2 be randomly, independently and uniformly chosen in $\mathcal{Z}_{R2^n}$, and set $x \leftarrow x_1 +_{R2^n} x_2$. Note that $f_R(x) = f_R(x_1) + f_R(x_2) + \zeta$ where $\zeta \in \{0, 1\}$, i.e. f_R is almost a linear function of its inputs. The Δ-*approximate linear consistency test* is

$$\text{“Is } P_R(x) \approx_\Delta P_R(x_1) + P_R(x_2)\text{?”},$$

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

Our main theorem with respect to the approximate self-tester for f_R is that the linear consistency error gives good bounds on $\text{apperr}(f_R, P_R, U_{\mathcal{Z}_{R2^n}}, \Delta_2)$: there are constants $0 < \psi < 1$ and $\psi' > 1$ such that $\text{apperr}(f_R, P_R, U_{\mathcal{Z}_{R2^n}}, \Delta_1)$ is at least ψ times the approximate linear consistency error, and that $\text{apperr}(f_R, P_R, U_{\mathcal{Z}_{R2^n}}, \Delta_2)$ is at most ψ' times the approximate linear consistency error. Thus, we can *indirectly* approximate $\text{apperr}(f_R, P_R, U_{\mathcal{Z}_{R2^n}}, \Delta_2)$ by instead estimating the approximate linear consistency error.

Program Quotient Function Approximate Self-Test (n, R, β)

```
$N = O(\ln(1/\beta))$
$t \leftarrow 0$
Do for $m = 1, \ldots, N$
  Call **Quotient_Linear_Test** $(n, R, ans, 3\Delta_1 + 1)$
  $t \leftarrow t + ans$
If $t/N > 1/256$ then "FAIL"
```

Quotient_Linear_Test (n, R, ans)

```
$ans \leftarrow 0$
Choose $x \in_{\mathcal{U}} \mathcal{Z}_{R2^n}$
Call **Random_Split**$(R2^n, x, x_1, x_2, c)$
If $|P(x_1, R) + P(x_2, R) - P(x, R) - c \cdot 2^n| \geq 3\Delta_1 + 1$ then $ans \leftarrow 1$
```

Theorem 13 *For* $\Delta_2 \geq 18\Delta_1 + 6$, *the above program is an* $(1/768, 1/8, \Delta_1, \Delta_2)$-*approximate self-testing program for the quotient function with any* R.

Proof: Corollary of Theorem 14 from the subsection on generic approximate self-testing. ■

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

4.2.2 Generic Approximately Linear Self-Testing

In this section, we describe a generalization of the quotient function approximate self-tester to functions f mapping a group G into $\mathcal{Z}$. In all cases, the resulting approximate self-testing program is extremely simple to code, different and efficient.

Let G be a finite group with group operation $+_G$. For $y \in G$, let y^{-1} denote the inverse of y. Let $f : G \rightarrow \mathcal{Z}$ be a function. Intuitively, f is hard to compute compared to either $+_G$ or $+$.

Let $\mathcal{U}_G$ be the uniform probability distribution on G. We say that f has the Δ-*approximate linearity property* if:

(1) It is easy to choose random elements of G according to $\mathcal{U}_G$.

(2) F_{applin} is an easily computable function with the property that, for any pair $x_1, x_2 \in G$, $F_{\text{applin}}(x_1, x_2) \in \mathcal{Z}$ and furthermore $f(x_1 +_G x_2) \approx_\Delta f(x_1) + f(x_2) + F_{\text{applin}}(x_1, x_2)$.

Note that if $f(x_1) \approx_{\Delta_1} g(x_1)$ and $f(x_2) \approx_{\Delta_1} g(x_2)$ then $f(x_1 +_G x_2) \approx_{2\Delta_1+\Delta} g(x_1)+g(x_2)+F_{\text{applin}}(x_1, x_2)$.

The approximate linearity property is a special case of approximate random self-reducibility.

Let P be a program that supposedly computes f such that, for all $y \in G$, $P(y) \in \mathcal{Z}$. Let $\Delta' \leftarrow \Delta + 3\Delta_1$, and $\Delta_2 \leftarrow 6\Delta'$. Then Generic Approximate Self-Testing Program is an $(\epsilon^2/12, \epsilon, \Delta_1, \Delta_2)$-approximate self-testing program for f with respect to $\mathcal{U}_G$. Note that the quotient function is 1-approximately linear. The approximate self-tester for the quotient function is based on Generic Approximate Self-Testing Program, where $G = \mathcal{Z}_{R2^n}$ with addition mod $R2^n$ as the group operation.

Generic Approximate Self-Testing Program $(\epsilon, \beta, \Delta')$

```
$N \leftarrow \frac{16}{\epsilon^2}\ln(2/\beta)$
$t \leftarrow 0$
Do for $m = 1, \ldots, N$
   Call **Generic_Linear_Test** $(ans, \Delta')$
   $t \leftarrow t + ans$
If $t/N > \epsilon^2/4$ then "FAIL" else "PASS"
```

Generic_Linear_Test (ans, Δ')

```
randomly choose $x_1 \in G$ according to $\mathcal{U}_G$.
randomly choose $x_2 \in G$ according to $\mathcal{U}_G$.
If $P(x_1 +_G x_2) \approx_{\Delta'} P(x_1) + P(x_2) + F_{\text{applin}}(x_1, x_2)$ then $ans \leftarrow 1$ else $ans \leftarrow 0$
```

We introduce some notation and provide motivation for why the approximate self-testers work. For each $y \in G$, define the *discrepancy of y* to be

$$\text{disc}(y) = f(y) - P(y).$$

Because of the approximate linearity property, part (2), and because the approximate self-testing program computes $F_{\text{applin}}(x_1, x_2)$ correctly on its own, with $\Delta' \leftarrow \Delta + 3\Delta_1$, we have that $P(x_1 +_G x_2) \approx_{\Delta'} P(x_1) + P(x_2) + F_{\text{applin}}(x_1, x_2)$ implies

$$\text{disc}(x_1 +_G x_2) \approx_{\Delta'} \text{disc}(x_1) + \text{disc}(x_2).$$

Theorem 14 *If f has the Δ-approximate linearity property, then the Generic Self-Testing Program is $(\epsilon^2/12, \epsilon, \Delta_1, \Delta_2)$-approximate self-testing for $(f, \mathcal{U}_G)$, for any $0 \le \epsilon \le 1/8$ where Δ_1 is arbitrary and Δ_2 is related to Δ_1 via the following: $\Delta' = \Delta + 3\Delta_1$, and $\Delta_2 = 6\Delta'$.*

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

- $\epsilon' = \Pr[|\text{disc}(x_1 +_G x_2) - \text{disc}(x_1) - \text{disc}(x_2)| \geq \Delta']$ when x_1 and x_2 are randomly and independently chosen according to $\mathcal{U}_G$.
- $\psi_1 = \Pr[|\text{disc}(y)| \geq \Delta_1]$ when y is randomly chosen in G according to $\mathcal{U}_G$.
- $\psi_2 = \Pr[|\text{disc}(y)| \geq \Delta_2]$ when y is randomly chosen in G according to $\mathcal{U}_G$.

Since $\Delta_2 \geq \Delta_1$, $\psi_1 \geq \psi_2$.

Uncapitalized letters from the end of the alphabet denote elements chosen randomly from G according to $\mathcal{U}_G$, e.g. x,y, z, whereas uncapitalized letters from the beginning of the alphabet denote fixed elements of G, e.g. a, b, c. $+_G$ denotes addition modulo $R2^n$.

Theorem 15 $\epsilon' \geq \psi_2^2$.

Before giving the proof of this theorem, we prove some intermediate lemmas. We assume that $\epsilon' < 1/64$ in Lemmas 31, 32, and 33. Let δ satisfy the equality $\delta^2 = \epsilon'$. Because $\epsilon' < 1/64$, $\delta < 1/8$.

Lemma 31 *$\forall a \in G$, $\exists a' \in \mathcal{Z}$ such that* $\Pr_{x \in \mathcal{U}_G}[\text{disc}(x +_G a) \approx_{2\Delta'} \text{disc}(x) + a' + \zeta] \geq 1 - 2\delta$.

Proof: [of Lemma 31] Let $S = \{a | \Pr_{x \in \mathcal{U}_G}[\text{disc}(x +_G a) \approx_{\Delta'} \text{disc}(x) + \text{disc}(a)] \geq 1 - \delta$. By a simple counting argument, $|S|/|G| \geq 1 - \delta$. For all $a \in S$, set $a' \leftarrow \text{disc}(a)$.

If $a = b +_G c$, and $b, c \in S$, let $a' \leftarrow b' + c'$ (if more than one pair of elements of S sum to a, then pick one pair arbitrarily). Then

$$\begin{aligned}\Pr_{x \in \mathcal{U}_G}[\text{disc}(x +_G b +_G c) &\approx_{\Delta'} \text{disc}(x +_G b) + c' \\ &\approx_{\Delta'} \text{disc}(x) + b' + c' \\ &= \text{disc}(x) + a'] \geq 1 - 2\delta.\end{aligned}$$

Thus $\Pr_{x \in \mathcal{U}_G}[\text{disc}(x +_G a) \approx_{2\Delta'} \text{disc}(x) + a'] \geq 1 - 2\delta$.

Let $T = \{a | \Pr_{x \in \mathcal{U}_G}[\text{disc}(x +_G a) \approx_{2\Delta'} \text{disc}(x) - a'] \geq 1 - 2\delta\}$.

Now we claim that $T = G$ and Lemma 31 follows. To prove the claim, it is enough to show that for $a \in G$, there exists an $a_1, a_2 \in S$ such that $a_1 +_G a_2 = a$. Pick $a_1 \in G$

uniformly at random, and let $a_2 \leftarrow a -_G a_1$. a_2 is then also distributed uniformly in G. Since $\delta < 1/2$, both $a_1, a_2 \in S$ with probability at least $1 - 2\delta > 0$. Thus there exists some pair $a_1, a_2 \in S$ such that $a_1 +_G a_2 = a$. ■

Lemma 31 leads to the definition of the function h from G to $\mathcal{Z}$ defined as follows: For all $a \in G$, let $h(a) = a'$, where a' is the element of $\mathcal{Z}$ described in Lemma 31.

Lemma 32 *For all $a, b \in G$, $h(a +_G b) \approx_{6\Delta'} h(a) + h(b)$.*

Proof: [of Lemma 32]

$$\begin{aligned}\Pr_{x \in \mathcal{U}_G}[\mathrm{disc}(x) + h(a) + h(b) &\approx_{2\Delta'} \mathrm{disc}(x +_G a) + h(b) \\ &\approx_{2\Delta'} \mathrm{disc}(x +_G a +_G b) \\ &\approx_{2\Delta'} \mathrm{disc}(x) + h(a +_G b)] \geq 1 - 6\delta.\end{aligned}$$

Thus $\Pr_{x \in \mathcal{U}_G}[\mathrm{disc}(x) + h(a) + h(b) \approx_{6\Delta'} \mathrm{disc}(x) + h(a +_G b)] \geq 1 - 6\delta$. This probability is strictly greater than zero because $\delta < 1/6$, and thus $h(a +_G b) \approx_{6\Delta'} h(a) + h(b)$. ■

Lemma 33 *For all $a \in G$, $|h(a)| \leq 6\Delta'$.*

Proof: Let t_1 be such that $h(t_1) = \min_{x \in G} h(x)$ and t_2 be such that $h(t_2) = \max_{x \in G} h(x)$. Then by Lemma 32, $h(t_1 +_G t_1) \leq 2h(t_1) + 6\Delta'$. If $h(t_1) < -6\Delta'$, the minimality of $h(t_1)$ is contradicted so $h(t_1) \geq -6\Delta'$. Similarly $h(t_2) \leq 6\Delta'$. ■

Proof: [of Theorem 15] By the way function h was defined, the disc function $(\delta, 0)$-approximates $(h, \mathcal{U}_G)$. Thus by Lemma 33, P $(\delta, 6\Delta')$-approximates $(f, \mathcal{U}_G)$. ■

Theorem 16 *$\psi_1 \geq \epsilon'/3$.*

Proof: Because $\psi_1 = \Pr[|\mathrm{disc}(y)| \geq \Delta_1]$, $\Pr[|\mathrm{disc}(x_1 +_G x_2)|, |\mathrm{disc}(x_1)|, |\mathrm{disc}(x_2)| \leq \Delta_1] \geq (1 - 3\psi_1)$, and consequently $\epsilon' \leq \Pr[\mathrm{disc}(x_1 +_G x_2) \approx_{3\Delta_1 + 1} \mathrm{disc}(x_1) + \mathrm{disc}(x_2)] \geq 1 - 3\psi_1$ ■

Proof: [of Theorem 14]

Similar to proof of Theorem 4 (p. 36). ■

4.3 Open Question

The generic self-testing method given in this chapter works for any function with the linearity property which maps to an infinite cyclic group, as well as the quotient function. Is there a modification of this technique which gives a self-tester for any function with the linearity property that maps to a finite cyclic group?

Chapter 5

Libraries and Linear Algebra

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 [15, 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 [3, 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. 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.

Previously, [39, Kannan] provides elegant program result checkers for the problems of computing the determinant and rank of a matrix, but they are not efficient. Our self-correcting/testing pairs for determinant and rank are efficient, but they rely 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 or rank.

The results in this chapter were done in collaboration with Manuel Blum and Michael Luby [20, Blum Luby Rubinfeld 2], [21, Blum Luby Rubinfeld 3].

5.1 Definitions

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

Definition 5.1.1 (library) *Let c be a positive integer. A* library *is a family of functions $f^1,\ldots,f^c$ with a corresponding set of input universes $\mathcal{I}^1,\ldots,\mathcal{I}^c$. A* distribution set *for a library is a family $\mathcal{D}^1,\ldots,\mathcal{D}^c$, where $\mathcal{D}^i$ is an ensemble of distributions on inputs $\mathcal{I}_n^i$ 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 5.1.2 (library result checking) *A* result checking program *for f^1 with respect to a library $f^1,\ldots,f^c$ with input universes $\mathcal{I}^1,\ldots,\mathcal{I}^c$ is a probabilistic program $R_f^{P^1,\ldots,P^c}$ that on input $x \in \mathcal{I}^1$ and β makes calls to $P^1,\ldots,P^c$. $R_{f_1}^{P^1,\ldots,P^c}$ has the following properties:*

1. *If for all $i = 1,\ldots,c$ and for all $y \in \mathcal{I}^i$, $f^i(y) = P^i(y)$ then $R_{f_1}^{P^1,\ldots,P^c}$ outputs "PASS" with probability at least $1-\beta$.*
2. *If $f^1(x) \neq P^1(x)$ then $R_{f_1}^{P^1,\ldots,P^c}$ outputs "FAIL" with probability at least $1-\beta$.*

Definition 5.1.3 (library self-testing) *A* self-testing program *for a library $f^1,\ldots,f^c$ with input set $\mathcal{I}^1,\ldots,\mathcal{I}^c$, distribution set set $\mathcal{D}^1,\ldots,\mathcal{D}^c$, error set $\epsilon_1^1,\ldots,\epsilon_1^c$ and error set $\epsilon_2^1,\ldots,\epsilon_2^c$, where, for each $i = 1,\ldots,c$, $\epsilon_1^i < \epsilon_2^i$, 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. $T_{f^1,\ldots,f^c}^{P^1,\ldots,P^c}$ has the following properties:*

1. *If, for all $i = 1,\ldots,c$, $\text{error}(f^i,P^i,\mathcal{D}_n^i) \leq \epsilon_1^i$ then $T_{f^1,\ldots,f^c}^{P^1,\ldots,P^c}$ outputs "PASS" with probability at least $1-\beta$.*

2. *If, for some* $i = 1, \ldots, c$, $\mathrm{error}(f^i, P^i, \mathcal{D}_n^i) \geq \epsilon_2^i$ *then* $T_{f^1,\ldots,f^c}^{P^1,\ldots,P^c}$ *outputs "FAIL" with probability at least* $1 - \beta$.

DEFINITION 5.1.4 (library self-correcting) *A* self-correcting program *for* f^1 *with respect to a library* $f^1, \ldots, f^c$ *with input set* $\mathcal{I}^1, \ldots, \mathcal{I}^c$*, distribution set set* $\mathcal{D}^1, \ldots, \mathcal{D}^c$*, and error set* $\epsilon^1, \ldots, \epsilon^c$ *is a probabilistic program* $C_{f^1}^{P^1,\ldots,P^c}$ *that on input* n*,* $x \in \mathcal{I}_n^1$ *and* β *makes calls to* $P^1, \ldots, P^c$ *to compute* $C_{f^1}^{P^1,\ldots,P^c}(x)$*.* $C_{f^1}^{P^1,\ldots,P^c}$ *has the property that if, for all* $i = 1, \ldots, c$, $\mathrm{error}(f^i, P^i, D_n^i) \leq \epsilon^i$ *then, for all* $x \in \mathcal{I}_n^1$*,* $C_{f^1}^{P^1,\ldots,P^c}(x) = f^1(x)$ *with probability at least* $1 - \beta$*.*

DEFINITION 5.1.5 (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.* $T_{f^1,\ldots,f^c}^{P^1,\ldots,P^c}$ *is a self-testing program for the library with some input set* $\mathcal{I}^1, \ldots, \mathcal{I}^c$*, distribution set set* $\mathcal{D}^1, \ldots, \mathcal{D}^c$*, and pair of error sets* $\epsilon_1^1, \ldots, \epsilon_1^c$ *and* $\epsilon_2^1, \ldots, \epsilon_2^c$*.* $C_{f^1}^{P^1,\ldots,P^c}$ *is a self-correcting program for* f^1 *with respect to the library with the same input set* $\mathcal{I}^1, \ldots, \mathcal{I}^c$ *and distribution set* $\mathcal{D}^1, \ldots, \mathcal{D}^c$ *and with an error set* $\epsilon^1, \ldots, \epsilon^c$*, where for all* $i = 1, \ldots, c$, $0 \leq \epsilon_1^i < \epsilon_2^i \leq \epsilon^i < 1$*.*

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 running time 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 running 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.

5.2 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 section.

DEFINITION 5.2.1 (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* $\mathrm{rank}(A)$ *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* $M_{n\times n}[F]$.

5.2.1 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 $\mathrm{rank}(A)$.

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** referred to below is given in Chapter 2, page 14.

Specifications of Matrix_Mult Self-Correct(n, A, B, β):
If $\mathrm{error}(f, P, \mathcal{U}_{M_{n\times n}[F]} \times \mathcal{U}_{M_{n\times n}[F]}) \leq 1/8$ 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
```

Lemma 34 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 $O(T(n) + n^2 \ln(1/\beta))$ because the expected number of iterations until $C = A \cdot B$ is at most two. ■

The self-testing program for matrix multiplication program is different. 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.

5.2.2 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 [54, 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 [54, 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 35 **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 17 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 different. 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 $\text{error}(f, P, \mathcal{U}_{M^n_{n\times n}[F]})$. 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 $\text{error}(f, P, \mathcal{U}_{M^n_{n\times n}[F]}) \geq 1/8$ 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 inversion.

5.2.3 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 $\mathbf{MISC}(n, A, 3/4) =$ "NO" then $answer_i \leftarrow 0$
      Else
        $R \leftarrow$ Gen_Inv_Matrix$(n)$
        $R' \leftarrow \mathbf{MMSC}(n, A, R, 1/16)$
        $d_R \leftarrow P(R)$
        $d_{R'} \leftarrow P(R')$
        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 36 Determinant Self-Correct *meets the specifications.*

As was the case for the self-testing program for matrix inversion, the self-tester for determinant is different 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 of $\text{error}(f, P, \mathcal{U}_{M^n_{n\times n}[F]})$. 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.

5.2.4 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 5.2.2 (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 37 **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 $B \in M_{2n \times 2n}[F]$, $\mathcal{E}_{2n}[\{B\}] \leq 2\mathcal{D}_{2n}[\{B\}]$. From this and the assumption that $\text{error}(f, P, \mathcal{D}_{2n}) \leq \frac{1}{16}$ it follows that

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

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

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

Thus, for each i, $\Pr[answer_i = \text{rank}(A)] \geq \frac{11}{16} > \frac{1}{2}$. 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 different.

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

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

(2) If $\text{error}(f, P, \mathcal{D}_n) \geq 1/16$ 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 38 **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 $1 - \beta$. ■

Chapter 6

Result Checkers for Parallel Programs

Many result checkers in the sequential model of computation have been found for various types of problems. However, a user is unlikely to be willing to use a sequential result checker to verify the correctness of a result produced by a fast parallel algorithm. In this chapter, we extend the program result checking framework to the setting of checking parallel programs and find general techniques for designing such result checkers. For example, we find techniques for result checking programs which compute certain types of functions that have the property that they can be computed be computed "indirectly", by calling the program on another, related input. We also present techniques based on quickly reconstructing the computation of a simple sequential algorithm, on duality and on constant depth reducibility among problems. We find result checkers for many basic problems in parallel computation.

The difference in the complexity of solving a problem as compared to the complexity of result checking a problem is often very dramatic. For example, we show that there are P-complete problems (evaluating straight-line programs, linear programming) that have very fast (even constant depth) parallel result checkers. Integer GCD is not known to be in RNC, yet a logarithmic depth parallel result checker exists for it [2, Adleman Huang Kompella]. Maximum Matching is not known to be in NC (though it is in RNC), and it has a deterministic NC result checker. Multiplication, parity and majority all have lower bounds of $\Omega(\log n / \log\log n)$ depth, yet all have (completely different) constant depth result checkers.

The results in this chapter also appear in [57, Rubinfeld 1].

6.1 The Parallel Program Result Checking Model

A parallel result checker must satisfy all of the same requirements as a sequential result checker mentioned in Chapter 2, with the following added remarks which are specific to the parallel programming setting.

The parallel result checker is allowed to call the program as many times as desired at each parallel step.

Both the running time of the parallel checker *and* the number of processors used by the parallel checker are quantities that are of interest. The *incremental time* (parallel depth) of the result checker includes the parallel time required by the result checker and the time to write the inputs to and read outputs from the program, but does not include any of the running time of the program. The *incremental number of processors* includes the number of processors used by the parallel checker and the processors required to write the inputs to and read outputs from the program, but does not include any of the processors used by the program.

The *total running time* of the result checker includes the parallel running time of the result checker and that of the program when called as a subroutine. This is the actual time required to perform the check. The *total number of processors* used by the result checker includes the number of processors used by the result checker, and those used by the program. This is the actual number of processors required to perform the check.

In order to enforce the result checker to be quantifiably different from any program which computes the function, we make the following definitions: Suppose that any parallel program running on a particular parallel model of computation (i.e. EREW PRAM, CRCW PRAM) which compute the function f requires at least d depth. Suppose the number of processors required when computing f in depth d is p. We say a parallel result checker is *quantifiably different* if it either (1) the incremental time is $o(d)$ or (2) the incremental time is $O(d)$ and (simultaneously) the incremental number of processors is $o(p)$ on the same model of parallel computation. All of the result checkers are quantifiably different.

For efficiency purposes we would also like to minimize the total parallel running time and total number of processors. All of the result checkers call the program at most once on any computation path, so the total depth is *big oh* of the depth of the program being checked. Many of the result checkers have the property that the total number of processors used is *big oh* of the number of processors used by the program (e.g. sorting, parity).

When describing the total running time of a result checker, we will use $D(n)$ to refer to the depth of the program running on an input of size n, and $N(n)$ to refer to the number of processors used by the program when running on an input of size n. None of the result checkers described here ever needs to call the program more than once along any computation path.

In all of the examples, the result checker first calls the program on the input which is being checked.

All of the examples in this chapter are written for the arbitrary and priority CRCW PRAM models.

6.2 Simple Parallel Result Checkers

Clearly the existence of a fast sequential result checker for a problem does not imply the existence of a fast or efficient parallel result checker because of the difficulty of parallelizing the result checker. In fact, some of the major ideas used in checking sequential programs do not seem to be as applicable in checking parallel programs. For example, one of the ideas proposed in [15, Blum] for the sequential checking of decision problems is the idea of reducing a search problem to a decision problem. This gives a simple proof of the correctness of a 'yes' instance. Results in [52, Mulmuley Vazirani Vazirani] show that there are techniques for reducing a search problem to a decision problem for weighted search problems. However results in [44, Karp Upfal Wigderson 2] about the difficulty of reducing search to decision in NC indicate that in general this idea could be difficult to utilize in parallel.

There are many problems for which the sequential result checking algorithm can be converted to a parallel result checking algorithm in a straightforward manner. For example, the deterministic result checker algorithm for sorting presented in Chapter 2 can be easily implemented in constant depth with $O(n)$ processors.

A sequential result checker for a program that finds the rank of a matrix (over a finite field) is described in [39, Kannan]. This result checker uses the ideas of interactive proofs to check that the program is correct by making sure that it is consistent with itself. By slightly changing it, the sequential result checker can be parallelized to run in $O(\log n)$ depth. The sequential result checker for Integer GCD [2, Adleman Huang Kompella] uses ideas from interactive proofs and can be made into a parallel result checker. It runs in $O(\log n)$ depth, with n processors.

Another example of a function for which there is a simpel result checker is that of the maximal independent set problem: simply verify that the output is an independent set, and then verify that it is maximal. This can be done in constant time with $O(n + m)$ processors where n is the number of nodes and m is the number of edges. Furthermore, the checker is deterministic, and no other calls to the program are made. In contrast, the best known algorithms for this problem use at least logarithmic parallel time [36, Goldberg Spencer], [5, Alon Babai Itai], [49, Luby].

6.3 Computability by Random Inputs

In Chapter 3, [20, Blum Luby Rubinfeld 2] we show that one can design result checkers for many functions that have the property of random self-reducibility - that the function can be computed by computing the function on one or more "random" instances. Here we show that often the property that a function can can be computed by computing the function on one or more "almost-random" instances can also be utilized in designing a result checker.

We concentrate on symmetric functions - functions on n bits whose output depends only on the *number* of 1's in the input. Thus, the value of the function can be computed indirectly by computing the function on a "shuffle" (random permutation) of the input bits. However, the techniques in this chapter are applicable to other functions as well. For example, using the technique in Section 6.3.2, the running time of the sequential self-tester for the matrix rank function given in [21, Blum Luby Rubinfeld 3] is dramatically improved in Chapter 5, page 72, [20, Blum Luby Rubinfeld 2].

The techniques in this section are based on testing the program on random inputs for which the answer is known, and then verifying that the program's answer on the particular input being checked is consistent with the program's answer on most other inputs.

6.3.1 Any Symmetric Function on n Bits

We give a result checker for any symmetric function:

Input: A list of input bits $\hat{a} = a_1, a_2, ..., a_n$, a table of values $\hat{t} = t_0, ..., t_n$.

Output: $b = t_i$ where $i = \Sigma_{1 \leq j \leq n} a_j$.

The majority, exactly i and parity functions are all examples of symmetric functions. As mentioned before, [14, Beame Hastad] show that $\Omega(\log n / \log\log n)$ depth is required to compute these functions. For these and other examples, no table is needed as input because the table can be computed in constant depth by the result checker.

Let P be the program that supposedly computes the symmetric function. P is checked by partitioning the inputs of size n into $n+1$ equivalence classes, where all inputs in a particular equivalence class contain the same number of 1's. Intuitively, the result checker verifies that P is correct on more than $1/2$ of the members of each equivalence class, and that the answer of P on the input in question is consistent with more than $1/2$ of the members within its own equivalence class. Therefore, even if the result checker cannot determine which equivalence class the input is in, it can verify that the answer of P on the input is correct. In the result checking algorithm, several random permutations of the input bits are made; [4, Ajtai] gives a way of doing this in constant depth.

Program Symmetric Function Result Check$(n, \hat{a}, \hat{t}, b, \beta)$

```
$k \leftarrow \log 1/\beta$
$b \leftarrow P(\hat{a})$
In parallel, compute $k$ random permutations $\pi_1, \ldots, \pi_k$ of $\{1, \ldots, n\}$
Phase 1: (Consistency with our input)
    In parallel, for $i = 1, \ldots k$
        If $P(\pi_i(\hat{a})) \neq b$ then output “FAIL” and halt
Phase 2: (Testing Correctness of most inputs)
    In parallel, for $j = 0, \ldots, n$
        In parallel, for $i = 1, \ldots, k$
            If $P(\pi_i(1^j 0^{n-j})) \neq t_j$ then output “FAIL” and halt.
Output “PASS”.
```

Proof: [of correctness of result checker] Clearly if P is correct on all inputs, the result checker will output “PASS”. Assume that P is incorrect on input $\hat{a}$, we show that the result checker outputs “FAIL” with probability $\geq 1 - \beta$. Let j be the number of ones in $\hat{a}$. Suppose that P is correct (and consequently differs from the output on $\hat{a}$) on $\geq 1/2$ the inputs with j ones. Then with probability $\geq 1 - \beta$, an input that is inconsistent with $\hat{a}$ is found in Phase 1. Suppose that the program errs on $\geq 1/2$ the inputs of size n with j ones. Then with probability $\geq 1 - \beta$, the j^{th} group of processors in Phase 2 finds that the program is buggy. Notice that by this argument the same k permutations can be used in Phase 1, and by every group of processors in Phase 2. ■

The incremental time is $O(1)$ and incremental number of processors is $O(C(n) + n^2)$ where $C(n)$ is the number of processors necessary to compute a random permutation in $O(1)$ parallel steps. The total time is $O(1 + D(n))$ and the total number of processors is $O(C(n) + nN(n))$.

6.3.2 Special Symmetric Functions

A factor of n in the number of processors can be saved when the symmetric function f is of a special type: Let $t_0, \ldots, t_n$ be the input table for problems of size n and $t'_0, \ldots, t'_{2n}$ be the input table for problems of size $2n$. We say that f is of this special type if there is an easily computable function $g(b, j)$ such that if $t'_i = b$ then $t_{i-j} = g(b, j)$.

Examples of such functions are parity, where $g(b, j) = b \oplus (j \bmod 2)$, and the unary to binary conversion function, where $g(b, j) = b - j$.

Program Special Symmetric Function Result Check$(n, \hat{a}, \hat{t}, b, \beta)$

```
$k \leftarrow O(\log 1/\beta)$
$b \leftarrow P(\hat{a})$
In parallel, compute $k$ random permutations $\pi_1, \ldots, \pi_k$ of $\{1, \ldots, 2n\}$
Phase 1: (Consistency with our input)
```

```
  In parallel, for $i = 1, \dots, k$:
    Uniformly and randomly pick $j \in [0, \dots, n]$
    Let $s$ be the string $a_1, ..., a_n, 1^j 0^{n-j}$
    $s' \leftarrow \pi_i(s)$
    If $b \neq g(P(s'), j)$, output "FAIL" and halt.
Phase 2: (Testing Correctness of most inputs)
  In parallel, for $i = 1, \dots, k$:
    Uniformly and randomly pick $j \in [0, \dots, 2n]$
    Create the string $s = 1^j 0^{2n-j}$
    $s' \leftarrow \pi_i(s)$
    If $P(s') \neq t'_j$, output "FAIL" and halt.
Output "PASS".
```

Proof: [of correctness of result checker (sketch)] If P is correct on all inputs, then clearly the result checker outputs "PASS". Let l be the number of 1's in $\hat{a}$ and suppose $b = P(\hat{a}) \neq t_l$. Let $\mathcal{D}$ be the probability distribution defined by (j, r), where j is chosen uniformly at random in $[0, \dots, 2n]$ and r is a random string of length $2n$ with j 1's. Let $\mathcal{D}'$ be the probability distribution defined by (j, r), where j is chosen uniformly at random in $[0, \dots, n]$ and r is a random string of length $2n$ with $j+l$ 1's. Let p be the probability that $P(r) \neq t'_j$ when (j, r) is chosen according to $\mathcal{D}$. If $p \geq 1/4$, then each execution of the loop in Phase 2 outputs "FAIL" and halts with probability at least $1/4$. Thus, the output is "FAIL" with probability at least $1 - \beta$. Now consider the case where $p \leq 1/4$. Let s' be a string of length $2n$ with $l+j$ 1's. By the properties of g, if $P(s') = t'_{l+j}$ then $g(P(s'), j) = t_l$. Furthermore, it can be shown that if $p \leq 1/4$ then $Pr[P(s') = t'_{l+j}] \geq 1/2$ when (j, s') is chosen according to $\mathcal{D}'$. These two facts imply that $Pr[g(P(s'), j) = t_l] \geq 1/2$ in each execution of the loop in Phase 1, and thus, since $b \neq t_l$, $Pr[g(P(s'), j) \neq b] \geq 1/2$ in each execution of the loop in Phase 1, in which case the output is "FAIL". Thus, the output is "FAIL" with probability at least $1 - \beta$. ■

The incremental time is $O(1)$ parallel steps and the incremental number of processors is $O(C(n) + n)$. The total time is $O(1 + D(n))$ and the total number of processors is $O(C(n) + N(n))$.

6.3.3 Randomly Self-Reducible, Linear and Smaller Self-Reducible Problems

If the program computes a function which is randomly self-reducible and either has the linearity property or is self-reducible to smaller inputs (see definitions on pages 31,46), the general techniques described in Chapter 3, [20, Blum Luby Rubinfeld 2] can be parallelized. This gives constant depth efficient result checkers for checking numerical problems such as integer multiplication, integer division, mod, modular multiplication, modular exponentiation, polynomial multiplication, squaring and matrix multiplication. The technique can also be used to give a result checker for parity that uses $O(1)$ incremental time and $O(n)$ incremental number of processors.

6.4 Consistency

Many problems have linear time sequential algorithms that are extremely simple and even possible to prove correct with formal verification methods. However, it is often the case that any parallel algorithm P for the same problem is necessarily radically different and more complex. Intuitively, a typical parallel result checker developed in this section calls P to reconstruct the computation steps of the extremely simple sequential algorithm, and then verifies the consistency between adjacent steps of the computation. This can be done very quickly, and independently of the algorithm actually used by P. This simple idea gives deterministic parallel result checkers for a surprising number problems. Some problems have result checkers that do not even need an additional call to P.

The *prefix sums* problem takes as input a list of elements $a_1, a_2, ..., a_n$, and outputs $(b_1, b_2, ..., b_n)$, where $b_i = a_1 \circ a_2 \circ a_3 \circ ... \circ a_i$ for an associative binary operator $\circ$. We assume that $\circ$ can be computed correctly by one processor in constant time. In order to verify that P computes the correct result, in parallel for $1 \leq i \leq n-1$, processor i checks that $b_i \circ a_{i+1} = b_{i+1}$. The incremental time is $O(1)$ and the incremental number of processors is n. The total depth is $O(D(n))$ with $O(N(n)+n)$ total processors. Note that the result checker makes no additional calls to P. A small variant of this result checker works for the *list ranking problem* as well in the same time and with the same number of processors.

The *sum* problem is similar to prefix sums, except that only b_n is output, and thus it is harder to check. The intermediate prefix answers $b_1, \ldots, b_{n-1}$ can be reconstructed as follows: In parallel for $1 \leq i \leq n$, group i of processors calls the program to compute $b_i = P(a_1, a_2, ..., a_i)$. Then processor i verifies that $b_i \circ a_{i+1} = b_{i+1}$. The incremental time is $O(1)$ and the incremental number of processors is $O(n^2)$. The total depth is $O(1)+D(n)$ with $O(n \times N(n))$ total processors.

The ideas in this result checker can be used for various problems, including *parity, addition of* n *numbers*, and can be modified to work for *straight-line programming* (when the variables are each set only once) and the expression evaluation problem. When the variables can be set more than once, the incremental time of straight-line programming is $O(\log n)$ using sorting.

A result checker for *integer multiplication* can also be constructed using this idea, where the input is 2 n-bit numbers a, b and the output is $a \times b$. The result checker algorithm is as follows: In parallel for $1 \leq i \leq n$, the i^{th} group of n processors asks the program to multiply a by the last i bits of b to get r_i. If the i^{th} least significant bit of b is a 0 then the result checker verifies that $r_i = r_{i-1}$, otherwise the result checker verifies that $r_i - r_{i-1} = a \times 2^i$. The incremental time is $O(1)$, and the incremental number of processors is $n \times A(n)$, where $A(n)$ is the number of processors required to do addition in constant depth. In [24, Chandra Fortune Lipton], it is shown that $A(n)$ is $O(ng^{-1}(n))$ for any strictly increasing primitive recursive function g. The total time is $O(D(n))$ with $O(n \times A(n) + n \times N(n))$ total processors.

Because of the following known results, all the checkers presented in this section are quantifiably different. The best known algorithm for prefix sums uses $O(n/\log n)$ processors and $O(\log n)$ depth ([47, Ladner Fischer],[31, Fich]). Multiplication can be done in $O(\log n/\log\log n)$ parallel depth, with $O(n^\epsilon)$ processors (for any ϵ). Any algorithm using only a polynomial number of processors for prefix sums, sum, parity and integer multiplication provably requires $\Omega(\log n/\log\log n)$ depth ([14, Beame Hastad]). Straight-line programming is P-complete.

6.4.1 Problems that can be solved using Dynamic Programming

Richard Karp has pointed out that the basic technique described in this section can be used to check any problem that can be solved sequentially using dynamic programming, regardless of the algorithm used by the program. By dynamic programming, we mean that there is some polynomial algorithm that computes the function on the whole set of inputs by evaluating the *same* function on smaller sets of inputs and somehow combining the results. This usually involves writing out the function on smaller sets of inputs in the form of a table. The idea behind the result checker is to call the program on *each* subproblem in parallel to fill in the table, and then verify that the entries of the table are consistent with each other. In most cases, this combination of results involves finding the minimum or maximum of a set of numbers. Since the minimum and maximum function can be computed in constant time, the incremental time is constant.

The following is an example:

Longest Common Subsequence

Input: Two strings $x = x_1x_2x_3...x_n$ and $y = y_1y_2y_3...y_n$.

Output: The length of the longest common subsequence of x and y.

Let $lcs(l,k)$ denote the length of the longest common subsequence of $x_lx_{l+1}...x_n$ and $y_ky_{k+1}...y_n$. Then the sequential dynamic programming algorithm used to solve the longest common subsequence problem builds up the table as follows: if $x_l = y_k$ then $lcs(l,k) = 1 + lcs(l+1,k+1)$, otherwise $lcs(l,k) = max\{lcs(l,k+1), lcs(l+1,k)\}$.

The algorithm for the result checker is:

```
Do for all $1 \leq l \leq n$
    Do for all $1 \leq k \leq n$
        $s_{lk} \leftarrow P(x_l...x_n, y_k...y_n)$
        Verify consistency:
            If $x_l = y_k$ verify that $s_{lk} = 1 + s_{l+1,k+1}$
            else verify that $s_{lk} = max\{s_{l,k+1}, s_{l+1,k}\}$
    If any of these verifications fail then output "FAIL" else output "PASS"
```

The incremental time is $O(1)$ and the incremental number of processors is $O(n^3)$. The total running time is $O(1 + D(n))$ with $O(n^3 + n^2 \times N(n))$ total processors.

6.4.2 All Pairs Shortest Path

Input: $n \times n$ adjacency matrix A, with a nonegative weight for each edge.

Output: Matrix $Dist$ specifying length of shortest path between every pair of nodes.

Program All Pairs Shortest Path Result Check(D, A):

```
    Do in parallel for each entry $D(u, v)$
        (1) check that $Dist(u, v) \leq A(u, v)$
        (2) check that for all $w$ that are neighbors of $v$, $Dist(u, w) + A(w, v) \geq$
         $Dist(u, v)$
        (3) check that $\exists w$ neigbor of $v$ such that $Dist(u, w) + A(w, v) = Dist(u, v)$
    If any of these checks fail then output "FAIL" else output "PASS"
```

Proof: [of correctness of result checker] It is clear that if the program is correct, the result checker will output "PASS". Suppose that the result checker outputs "PASS". Let $d(u, v)$ denote the correct shortest distance between u and v. We want to show that for all pairs (u, v), $Dist(u, v) = d(u, v)$.

Suppose for contradiction that there are nodes u, v such that $Dist(u, v) < d(u, v)$. Let u, v be nodes with $Dist(u, v) < d(u, v)$ such that v has the smallest possible index. Then because of step 3, there must be a w such that $Dist(u, w) < d(u, w)$ and w has smaller index that v. Therefore, for all u, v we have that $Dist(u, v) \geq d(u, v)$.

We will show by induction on the number of intermediate nodes along a shortest path between a pair of nodes that $Dist(u, v) = d(u, v)$.

Basis: The number of intermediate nodes visited when taking the shortest path from u to v is 0 (edge uv is the shortest path). Step 1 guarantees that $Dist(u, v) \leq A(u, v) = d(u, v)$.

Induction Step: Suppose that $Dist(u, v) = d(u, v)$ for all pairs (u, v) where there is a shortest path from u to v that visits i intermediate nodes. Consider pair (u, v) where there is a shortest path from u to v with $i + 1$ intermediate nodes, and let w be the last node along this path. Then, step 2 verifies that $Dist(u, w) + A(w, v) \geq Dist(u, v)$. We know $d(u, w) + A(w, v) = d(u, v)$. By the induction hypothesis, since there is a shortest path between u and w of length i, $Dist(u,w) = d(u,w)$. Thus $Dist(u,v) \leq A(w,v) + d(u,w) = d(u,v)$ and so $Dist(u,v) = d(u,v)$. ■

The incremental time is $O(1)$ and the incremental number of processors is $O(n^3)$. The total time is $D(n)+O(1)$ with $O(n^3)+N(n)$ total processors. Note that the result checker makes no extra calls.

6.5 Duality

When result checking an optimization problem, it is necessary to check that the solution is as good as is claimed, and that it is the best solution. Duality can sometimes be used to show the latter.

For example, to result check a program that does linear programming, the result checker needs only check that the optimal solution is feasible, and to call the program again on the dual problem (again making sure that it is feasible) to check that the solution to the original problem is the same (and therefore optimal). If the program claims that there is no solution or that the solution is unbounded, this can be verified symbolically using the program in [18, Blum Kannan Rubinfeld]. This problem is P-complete, so no fast parallel algorithm is known for it. However, it can be result checked in logarithmic time with only two calls to the program.

Another example is the following:

Maximum Matching

Input: Graph $G = (V, E)$

Output: k = the size of a maximum matching, and the edges in a maximum matching in G

No deterministic NC algorithm is known for this problem, but it is known to be in RNC ([43, Karp Upfal Wigderson], [52, Mulmuley Vazirani Vazirani]).

Result Checking Algorithm:(sketch)

The result checker first checks in parallel that no vertex is matched more than once and that the maximum matching is of size k. Then the algorithm in [40, Karloff] is used to find a proof that there is no matching of size $\geq k$. This proof will be an odd set cover of size k. Karloff's algorithm requires computing a maximal independent set. This computation can be checked using the result checker described earlier. Karloff's algorithm also calls a matching oracle on other problem instances. The result checker calls the matching program on these instances, and proceeds as if all of the answers are correct. If the output of his algorithm is an odd set cover of size $\neq k$, the result checker outputs "FAIL". Otherwise, the odd set cover of size k is verification that the maximum matching is of size k, and the result checker outputs "PASS".

The incremental time is $O(d^{MIS}(n))$ parallel steps and $O(p^{MIS}(n))$ processors, where $d^{MIS}(n)$ is the parallel depth and $p^{MIS}(n)$ is the number of processors required to find a maximal independent set in an n node graph. The total running time is $O(d^{MIS}(n)+D(n))$ with $O(n^3 \times N(n) + p^{MIS}(n))$ processors.

6.6 Constant Depth Reducible Functions

We can say something about the relationship among result checking problems that are AC^0 equivalent.

Proposition 39 *Let π_1, π_2 be two AC^0 equivalent computational problems. Then from any fast program result checker C_{π_1} for π_1, is is possible to construct a fast program result checker C_{π_2} for π_2.*

Proof: Similar to Beigel's trick described in [15, Blum]. We outline the proof for decision problems, but the general proof is similar. The idea is to construct a program result checker for π_2 by transforming it to an instance of π_1 and result checking that instance. Since the oracle program still only solves π_2, in order to get an oracle for π_1 on x, we use the reverse transformation on x into an instance of π_2, and call the oracle for π_2 on it. Since the transformation and the reverse transformation can be computed in AC^0, the depth of the result checker for π_2 will be at most a constant times the depth of the result checker for π_1. Since π_1 and π_2 are AC^0 equivalent, the fastest parallel program for each is related by a constant factor. Therefore, if π_1 is a fast program result checker, so is π_2. ■

More recently, in [39, Kannan] Section 2.3, Beigel's theorem was generalized to problems in the same *robust* complexity classes, and used to show that if two problems are equivalent under NC reductions, and if one has a result checker, then so does the other.

We have already shown two P-complete problems that are checkable in small depth: linear programming and straight line programming. In [39, Kannan] it is observed that since P-complete problems are all NC-reducible to each other, all P-complete problems are checkable in polylogarithmic depth. Moreover, a problem is presented for which programs can be written that run in small depth, but for which result checking the result is P-complete.

Chapter 7

Adaptive Programs and Cryptographic Settings

In the theory of program result checking introduced in [15, Blum], P is always assumed to be a fixed program, whose output on input x is a static function $P(x)$. This is not always the case, as there are programs whose behavior changes as they run, even though the functions that they supposedly compute remain fixed. For example, hardware errors may evolve over time depending on the previous inputs that the program has been run on, or, the software may be written such that running the program on certain inputs may have unintended side effects on the program's future behavior. This could occur in a program that stores tables of previously computed information in a permanent file to make subsequent processing more efficient.

We extend the theory to check a program P which returns a result on input x that may depend on previous questions asked of P. We call such a program that can modify itself and its subsequent computation an *adaptive program*. We call a result checker that works for such a program an *adaptive result checker*. The work in this chapter was done in collaboration with Manuel Blum and Michael Luby [19, Blum Luby Rubinfeld 1].

This model is a restriction of the model used in interactive proof systems of [37, Goldwasser Micali Rackoff], in which the role of the verifier is played by the result checker and the role of the prover is played by the program P. The restriction is that the verifier may only ask questions of the form "What is the value of $f(x)$?". All result checkers extract an interactive proof of correctness from the program P. Since we do not always know how to extract such an interactive proof from a single program P running on one machine, we allow one program that supposedly computes f on each of k noninteracting machines, where k is a parameter which we would like to minimize. This corresponds to a restriction of the multi-prover interactive proof systems model of [13, Ben Or Goldwasser Kilian Wigderson] where k is the number of provers. We design adaptive result checkers that work for a constant number of independent and noninteracting programs.

Because the program can change and become faulty at any time, testing is of no interest when the program is adaptive. Self-correcting will also be impossible, because there is no notion of having a program that has been certified to be usually correct.

We make the following definitions:

DEFINITION 7.0.1 (adaptive result checker) *Program R is an* adaptive result checker *for f if R is a result checker for f that also works with respect to adaptive programs that supposedly compute f.*

DEFINITION 7.0.2 (k-adaptive result checker) *Program R is a k-*adaptive result checker *if R is a result checker for k adaptive programs which do not communicate among themselves.*

An adaptive result checker is automatically a k-adaptive result checker, and a k-adaptive result checker is automatically a (non-adaptive) result checker.

Many result checkers that have been found are also adaptive result checkers. For example, it can easily be seen that the GCD checker in [2, Adleman Huang Kompella] and that all of the result checkers given in [17, Blum Kannan] are adaptive. Other result checkers do not work for an adaptive program. Examples of such result checkers are the ones in Chapter 3, [20, Blum Luby Rubinfeld 2], where adaptive programs can easily fool the result checkers. At the present time we see no way to convert such a result checker into an adaptive result checker. However, if more than one copy of the program exists, we show that result checkers based on the methods in Chapter 3 can work for adaptive programs.

Next suppose we are in the following cryptographic situation: A user wants to evaluate function f on input x using program P running on another machine. As in result checking, the user does not trust the program to be correct. The additional requirement is that the user wants to let the other machine know as little information as possible about x from the questions asked of the program P (for example, the user may want the program to be able to learn at most the input size). This is similar to the model introduced in [1, Abadi Feigenbaum Kilian] and later extended in [9, Beaver Feigenbaum] to allow using several non-communicating programs for the same function, except that here we do not trust the program to return correct answers. In addition, we only allow protocols which are restricted versions of [1, Abadi Feigenbaum Kilian] [9, Beaver Feigenbaum] where the result checker may only ask the program questions of the form "What is the value of $f(x)$?". We call a program that satisfies the above constraints a *private result checker.* As is the case for adaptive result checking, we consider the case where there is a program that supposedly computes f on each of several noninteracting machines.

We introduce some notation in order to define a private result checker:

Let k be the number of programs purporting to compute f, such that none of these programs can communicate with any other program, and let P_i be the program on the i^{th} machine for $1 \leq i \leq k$.

As in [1, Abadi Feigenbaum Kilian], we define L to be a function which we call the *leak function.* Intuitively, $L(x)$ is the amount of information leaked by the result checker to the programs on input x. An example is $L(x) = |x|$, i.e. the result checker leaks the length of x to the programs, but nothing more.

Let $CONV_i[x]$ denote the probability distribution of the variable representing the concatenation of the questions that C asks of P_i on input x and let $Pr_{CONV_i[x]}(y)$ denote the probability of the string y according to the distribution.

DEFINITION 7.0.3 ((k, L)-private) *A program C is* (k, L)-private *if for all k-tuples of programs $(P_1, ..., P_k)$, for all v, w such that $L(v) = L(w)$, for all $1 \leq i \leq k$ and all y, $Pr_{CONV_i[v]}(y) = Pr_{CONV_i[w]}(y)$.*

If a result checker C is (k, L)-private where L leaks a very small amount of information about x (e.g. the length of x), then with high probability C does not ask any of the programs $P_1, .., P_k$ to evaluate input x. This means that the usual way of defining a result checker to output "FAIL" on input x if $P(x) \neq f(x)$ is insufficient. We define a private result checker as follows:

DEFINITION 7.0.4 ((k, L)-private result checker) *A program C is a* (k, L)-private result checker *if C is (k, L)-private and on input x and β, outputs $C(x)$ satisfying the following conditions: (1) if $P_1, ..., P_k$ answer correctly on all inputs, $C(x) = f(x)$. (2) $Pr[C(x) = f(x)$ or $C(x) = $ "$FAIL''] \geq 1 - \beta$.*

Thus, C outputs the correct answer if all programs always compute f as they should, but on the other hand it is unlikely that they can fool C into outputting the wrong answer (with probability at most β).

DEFINITION 7.0.5 ((k, L)-private/adaptive result checker) *C is a* (k, L)-private/adaptive result checker *if it is a k-adaptive result checker and a (k, L)-private result checker.*

We ask that the adaptive and private result checkers be *different*, and as efficient as possible. A (k, L)-private result checker or a k-adaptive result checker is $f(n)$-efficient if the total work done *by all k programs* and the result checker is $f(n)$ multiplied by the running time of the program.

We present general techniques for constructing simple to program and efficient (k, L)-private and 2-adaptive result checkers, and where L is a function that does not leak much about the input (for example only the size of the input), for a variety of numerical problems. The result checkers given in this paper are all based on the algorithms given in Chapter 3, though the proofs are different. They apply to integer multiplication, the mod function, modular multiplication, modular exponentiation, integer division, and polynomial and matrix multiplication over finite fields. For all problems, the result checker algorithms are both efficient and different. Furthermore, the result checker algorithms consists of the execution of the following basic operations at most a logarithmic number of times in a prescribed order: (1) calls to P on random instances of the problem; (2) additions; (3) comparisons.

7.1 Related Work

[Abadi Feigenbaum Kilian] show that there is not likely to be a $(1, |x|)$-private result checker for SAT that runs in polynomial time (not including the time required by the oracle). [Beaver Feigenbaum] describe how to compute any function privately with $O(|x|)$ oracles that are trusted not to err, however the oracles are not restricted to answer questions of the form "What is the value of $f(x)$?". [Beaver Feigenbaum Kilian Rogaway] later improved this result to show that it can be done with $O(|x|/\log|x|)$ oracles.

[9, Beaver Feigenbaum] [48, Lipton] show that any function that is a polynomial of degree d over a finite field is d-random-self-reducible. Thus, one can get a $(O(d), L)$-private result checker for the function, where L is the description of the finite field, under the following conditions: (1) the program is not adaptive and (2) the program is already known to be correct on a large fraction of inputs in the finite field.

In [13, Ben Or Goldwasser Kilian Wigderson] [33, Fortnow Rompel Sipser], there is a general technique for turning any result checker into a 2-adaptive result checker. This technique can actually be used for many of the result checkers. However, it requires a quadratic blowup in the number of calls made. Thus, if the number of calls made to the program is not constant, the extra work done by their technique is not of the same time order. For example, we give a way of converting one of the result checking techniques in Chapter 3, which makes $O(\log n)$ calls to the program, into an adaptive result checker which is of the same efficiency as the original result checker. The techniques of [13, Ben Or Goldwasser Kilian Wigderson] [33, Fortnow Rompel Sipser] yield an adaptive result checker that is slower than the original result checker by a multiplicative factor of $O(\log n)$.

In [13, Ben Or Goldwasser Kilian Wigderson], there is a general technique for turning any k-adaptive result checker into a 2-adaptive result checker. This technique requires an additional $O(k)$ multiplicative factor in the number of calls made.

Previous to our work, [38, Kaminski] introduced a result checker for integer and polynomial multiplication based on computing the result of the program mod small special numbers. This result checker trivially works for an adaptive program as well, because it makes no extra calls to the program. Independently of our work, [2, Adleman Huang Kompella] describe a result checker for multiplication in the same spirit but different than [38, Kaminski] which also makes no calls to the program. Also previous to our work, [34, Freivalds] introduced a result checker for matrix multiplication which does not call the program.

7.2 Private/Adaptive Checker

We first show how to construct a different and efficient result checker that is both adaptive *and* private for any function that has the linearity property. The result checker is an adaptation of the self-testing/correcting pair using the method based on linearity given in Chapter 3.

Theorem 17 *Any function which is computable by random homomorphisms has efficient and different 2-adaptive and (k, L)-private/adaptive result checkers, for constant $k = max\{4, c_1 + 1\}$, where $L(x) = G$ (G is the underlying group).*

Proof: [of Theorem 17 (idea)] [13, Ben-Or Goldwasser Kilian Wigderson] [33, Fortnow Rompel Sipser] show how to transform any result checker into a 2-adaptive result checker by simply running the original result checking protocol with the first program. If the original result checker would have accepted, a random question asked of the first program is chosen, and is also asked of the second program. If the second program gives the same answer as the first program, then the adaptive result checker returns "PASS". Otherwise, if the original result checker would have returned "FAIL", or if the second program answers differently than the first, the adaptive result checker returns "FAIL". An adaptation of the algorithm for self-testing/correcting based on linearity in Chapter 3 combined with the technique of [Fortnow Rompel Sipser] proves Theorem 2: The idea is that since every verification made by the checker involves program calls made on inputs that are uniformly distributed, though not independent from each other, each call can be made to a different program. Thus, each program sees a uniformly distributed input which is independent from the input being checked. Of these calls, one is chosen at random and asked of yet another program to verify that the same answer is given. ■

Some examples of programs that can be checked using this method are:

Problem	#Progs.	L	Without P	Total
Mod Mult. $f(x, y, R) = x \cdot y \bmod R$	6	$\|x\|, \|y\|, R$	n	$M(n)$
Mod $f(x, R) = x \bmod R$	4	R	n	$M(n)$
Integer Div. $f(x, R) = (x \text{ div } R, x \bmod R)$	4	$\|x\|, R$	n	$M(n)$
Mod Exp.,ϕ $f(x, y, R) = x^y \bmod R$	5	$x, \|y\|, R$	n	$M(n)$

When applied to checkers based on bootstrap self-testing, this method causes a slowdown by an $O(\log n)$ multiplicative factor. We give the following efficient way of constructing a private-adaptive checker from checkers that are based on bootstrap self-testing as described in Chapter 3.

Algorithm:

The algorithm is designed to run asking questions of programs $P_1, \ldots, P_{c_1}$ running on non-communicating machines. We describe the algorithm as if the questions are asked and immediately answered. However, the actual order of the questions is as follows: Let Q_i be the set of questions asked of P_i. The questions in Q_i are asked in a randomly permuted order, and then the verifications are done once all of the answers have been given. This can be done because none of the questions asked depend on results of previous questions.

We make the convention that if any call to one of the subroutines returns "FAIL" then the entire result checker program outputs "FAIL" and halts immediately.

The inputs to the private-adaptive result checker are $n \in \mathcal{N}$, $x \in \mathcal{I}_n$.

Program Private-Adaptive Check(l, x, β)

```
$N \leftarrow O(c \ln(1/\beta))$
Do for $m = 1, \ldots, N$
    For $i = 1, \ldots, l$, call **Private-Adaptive_Gen_Rec_ST**$(i)$
    $answer$ =**Private-Adaptive_Gen_SC**$(l, \hat{x})$
Output ($answer$,"PASS")
```

Subroutine Private-Adaptive_Gen_Rec_ST(n)

```
Choose $x \in_{\mathcal{U}} \mathcal{I}_n$
If $n = 1$ then:
  Compute $f(x)$ directly
  If $\exists i, 1 \leq i \leq c_2$, such that $f(x) \neq P_i(x)$ then output "FAIL" and halt
Else $n > 1$ then:
  Randomly generate $a_1, \ldots, a_{c_2}$ from $x$
  For $k = 1, \ldots, c_2$
    $y_k \leftarrow$ **Private-Adaptive_Gen_SC**$(n-1, a_k, \frac{1}{16{c_2}^2})$
  If $\exists i, 1 \leq i \leq c_2$, such that $F_{\text{smaller}}(x, a_1, \ldots, a_{c_2}, y_1, \ldots, y_{c_2}) \neq P_i(x)(*)$
      then output "FAIL" and halt.
```

Subroutine Private-Adaptive_Gen_SC(n, x, β)

```
Randomly generate $a_1, \ldots, a_{c_1}$ based on $x$
For $i = 1, \ldots, c_1$, $\alpha_i \leftarrow P_i(a_i)(**)$
$answer \leftarrow F_{\text{random}}(x, a_1, \ldots, a_{c_1}, \alpha_1, \ldots, \alpha_{c_1})$
Output $answer$
```

This result checker uses the method of self-testing based on bootstrapping described in Chapter 3, which tests the program on successively larger ranges, bootstrapping on the fact that the smaller ranges have already been tested. Since testing has no meaning for an adaptive program, the proofs given in Chapter 3 do not work in this setting. In fact, a naive implementation of the bootstrap protocol described in Chapter 3 can be fooled by c_1 adaptive programs, because the adaptive program can figure out where the result checker is in the computation by the questions asked of it, and lie accordingly. The above protocol overcomes this by asking the questions on each machine in a random order. Using the techniques of [13, Ben Or Goldwasser Kilian Wigderson], one can simply transform the bootstrap result checker into a 2-adaptive result checker, with an additional cost of $O(\log n)$ multiplicative overhead in the running time over the original result checker. On the other hand, the adaptive result checker presented here is as efficient as the original bootstrap result checker.

Theorem 18 *Any function which is computable by random inputs and computable by smaller inputs has a different and $T(x)$-efficient (c_1, L)-private/adaptive result checker, where $T(x) = L(x)$ is the* size *of x.*

Proof: [of Theorem 18] The intuition behind why this result checker works in the adaptive setting is that the questions are being sent to c_1 adaptive programs in such a way that the programs do not know whether the question was generated at random in line (*) of Private-Adaptive_Gen_Recursive_ST or whether the question was generated at random in line (**) of Private-Adaptive_Gen_SC. We show that this is enough to get a c_1-adaptive result checker. Let m be the total number of questions asked to program P_i and let $q_1, \ldots, q_m$ be the questions asked of P_i. Each program receives a random permutation of the questions $(q_1, q_2, \ldots, q_m)$. One can easily verify that the distribution of the questions is the same for all inputs of the same size, showing that the result checker is (c_1, L)-private. One can also easily verify that $q_1, \ldots, q_m$ are independently and uniformly distributed (but not identically distributed, there are a subset of questions that are uniformly distributed in $\mathcal{I}^r$ for each $r = 1, \ldots, l$.) Some of the questions are generated in Private-Adaptive_Gen_Recursive_ST, and some of the questions are generated in Private-Adaptive_Gen_SC in order to verify the questions in Private-Adaptive_Gen_Recursive_ST. Notice that the questions asked in Private-Adaptive_Gen_Recursive_ST are verified by computing them from questions that are of a smaller size. Let r be the smallest sized question on which *any* program errs and let P_i be a program that errs on an input of size r. Since the questions are asked in a randomly permuted order, with probability p where $1/(c_2 + 2) \leq p \leq 1/(c_2 + 1)$, the question was generated in Private-Adaptive_Gen_Recursive_ST rather than Private-Adaptive_Gen_SC. This is because Private-Adaptive_Gen_Recursive_ST only makes one call to P_i on inputs of size r, whereas Private-Adaptive_Gen_SC is called c_2 times on inputs of size r ($c_2 + 1$ times on inputs of size l) and makes one call to P_i on an input of size r each time that it is called. The program P_i cannot tell which subroutine generated the questions of size r because they are asked in a random order. If P_i errs on a question generated by Private-Adaptive_Gen_Recursive_ST, then if $r > 1$, since the question is being verified with smaller inputs, all of which are correct (by choice of r), the mistake is caught. Otherwise, if $r = 1$, the question is being verified by computation done by the result checker, and the mistake is caught.

To decrease the probability of error to $\leq \beta$, run the protocol $O(\log 1/\beta)$ times sequentially. If *answer* is always the same, output *answer*, otherwise output "FAIL". ■

This outline can be used to develop different and efficient adaptive and private result checker for the following problems:

Problem	#Progs.	L	Without P	Total
Int. Mult.	4	$\|x\|, \|y\|$	n	$M(n)$
Poly. Mult.	4	$deg(p), deg(q)$	n	$M(n)$
Mod Exp., no ϕ	5	$\|x\|, \|y\|, R$	$n \ln^4 n$	$M(n) \ln^3 n$
Matrix Mult.	4	n	n	$M(n)$

Using the method of [13, Ben Or Goldwasser Kilian Wigderson], one can convert all of the above adaptive result checkers into 2-adaptive result checkers. However, the stated privacy constraints will no longer be satisfied.

7.3 Open Questions

By the results of [33, Fortnow Rompel Sipser], any checker can be converted into a 2-adaptive result checker. A question that arises naturally is whether a result checker can in general be converted into a 1-adaptive result checker, as opposed to 2-adaptive. Since there is a complete language in EXPTIME that has a result checker [8, Babai Fortnow Lund], there is no general technique that converts any result checker into a 1-adaptive result checker unless EXPTIME=PSPACE. To see this, suppose there is such a general technique and consider the result checker for the complete language in EXPTIME. Now, because we can supposedly convert this result checker into a 1-adaptive result checker, there is an interactive proof for the language. Then by the result of [29, Feldman] the language must be in PSPACE.

Since there is probably no general technique for converting a result checker into a 1-adaptive result checker, it would be interesting to charactize which problems do have adaptive result checkers.

Another interesting question is under what conditions on L it is true that a $(1, L)$-private result checker is always a 1-adaptive result checker?

[33, Fortnow Rompel Sipser] have shown a technique by which any result checker can be made into a 2-adaptive result checker. Is there a more efficient technique which does the same thing?

Chapter 8

Batch Result Checking

Though many programmers are willing to spend some time overhead in order to verify that their programs give correct answers, for some applications, where efficiency is crucial, even a constant multiplicative time overhead makes result checking undesirable. In this chapter, we define a variant model of result checking, called *batch result checking*: Often greater efficiency can be achieved if the user does not need to know immediately whether the program gives the correct result. In this case, the result checker can wait until the program has been called on several inputs and check that the program is correct on all of the inputs at once. Batch result checking can allow greater efficiency, and we give examples of functions for which batch result checking allows one to reduce the overhead of the result checking process to the point where it is arbitrarily small.

A batch result checker is a result checker that checks that the program is correct on several inputs at once, and outputs "FAIL" if the program is incorrect on *any* of the inputs:

Definition 8.0.1 (probabilistic batch program result checker) *A probabilistic batch program result checker for f is a probabilistic oracle program R_f which is used to verify,* for any program P *that supposedly evaluates f, that P outputs the correct answer on several given inputs in the following sense. On given inputs $x_1, \ldots, x_m$ and confidence parameter β, R_f^P has the following properties:*

1. *If $\exists i$ such that $P(x_i) \neq f(x_i)$ then R_f^P outputs "FAIL" (with probability $\geq 1 - \beta$).*
2. *If P is a correct program for every input then R_f^P outputs "PASS" (with probability $\geq 1 - \beta$).*

Often a batch result checker can be made more efficient. For example, recall that a self-testing/correcting pair can be used to construct a result checker: use the self-tester to test the program. If the program fails the test, output "FAIL" and halt, and if the program passes, use the self-corrector to compute the correct result for the input being checked with high confidence, and compare the "correct" result to the output of the program on that input. Suppose the self-tester requires total time T, and the self-corrector requires total time S, then the incremental time is $T + S$. To check m inputs, rather than running the result checker m times, for a total running time of $m(T + S)$, the tester need only be run once, giving a total running time of $T + mS$. Since T time is usually much larger than S (for example, the self-tester for the mod function makes several hundred calls to the program while the self-corrector makes fewer than 20), this savings can be quite significant.

However, the following technique can be used to reduce the multiplicative overhead to arbitrarily close to 1 for any function with the linearity property (for definition see page 31). We present the specific batch result checker that results from applying the technique to the mod R function. The technique is based on the idea of Freivalds [34, Freivalds] used in the result checker for matrix multiplication (see Section 2.2.2, page 14). Freivalds' idea was also used in a similar setting by [30, Fiat Naor] where many modular exponentiation computations are verified by doing very few modular exponentiation computations.

Program Mod Function Batch_Checker$(n, R, x_1, \ldots, x_m, \beta)$

```
For $i = 1, ..., O(log(1/\beta))$ do:
  Randomly generate $m$-bit 0/1 vector $\alpha$
  $sumin \leftarrow 0$
  $sumout \leftarrow 0$
  For $i = 1, ..., m$ do
    $sumin \leftarrow sumin +_{R2^n} \alpha_i \cdot x_i$
    $sumout \leftarrow sumout +_R \alpha_i \cdot P(x_i)$
  Verify that $sumout = P(sumin)$
  Call $Mod\_Result\_Checker(sumin, 1/4)$
  If verification fails or checker returns "FAIL" then
  output "FAIL" and stop.
Output "PASS".
```

Proof: [of correctness of batch checker] Since the mod R function is linear, $f(\sum_{R2^n} \alpha_i x_i) = \sum_R \alpha_i f(x_i)$. Thus, if P is always correct, the checker outputs "PASS". If there is an $i \in [1 \ldots m]$ such that $P(x_i) \neq f(x_i)$, then with probability at most 1/2, $\sum_R \alpha_i P(x_i) = \sum_R \alpha_i f(x_i)$. Suppose $\sum_R \alpha_i P(x_i) \neq \sum_R \alpha_i f(x_i)$. If the verification that $sumout = P(sumin)$ passes ($\sum_R \alpha_i P(x_i) \neq P(\sum_{R2^n} \alpha_i x_i)$), we know that $P(\sum_{R2^n} \alpha_i x_i) \neq f(\sum_{R2^n} \alpha_i x_i)$. Then the call to **Mod_Result_Checker** passes with probability at most 1/4. Thus the batch checker outputs "FAIL" with probability at least 1/4 after one iteration. The proof follows from Proposition 1. ■

Let $T(n)$ be the running time of P on inputs of size n. From the self-testing/correcting pair for the mod function, a result checker for the mod function can be designed such that

Mod_Result_Checker requires at most $c \cdot T(n)$ total time for constant c. The total work done by the batch checker is $(c + m)T(n) + O(n \cdot m)$. Since the program is called on all of the m x_i's regardless of whether any result checking is done, the multiplicative running time overhead required by batch result checking is less than $1 + \frac{c+1}{m}$.

Chapter 9

Conclusions

We have presented a framework and given several techniques for writing self-testing/correcting programs. We have extended the result checking model to apply to several different settings, including parallel programs and programs that are adaptive. It is now of interest to characterize the problems that have program result checkers, self-testers and self-correctors in each of these settings.

We stress that since a program result checker for a function can easily be converted into a self-tester for that function, and since many types of functions have program result checkers, functions that have self-testers do not necessarily have any special structure. However, the same cannot be said for functions that have self-correctors: All of the known self-correctors rely on the property of random self-reducibility. This motivates the study of which functions are random self-reducible. The definition of random self-reducibility given in Chapter 3 differs from the standard definition (see for example [28, Feigenbaum Kannan Nisan]) in that it requires the reduction to be faster than any program for the original function, whereas the latter only requires that the reduction be in polynomial time. For the purposes of this discussion, we refer to the latter notion of random self-reducibility as polynomial time random self-reducibility. [28, Feigenbaum Kannan Nisan] have shown that random boolean functions are not polynomial time random self-reducible, and that if a function is polynomial time 2-random self-reducible (for definition see [28]), then the function can be computed nonuniformly in nondeterministic polynomial time. No such results are known for random self-reducibility, because for random self-reducibility the only requirement is that the computation of the random self-reduction be faster than the computation of the function itself. Thus for functions that do not have polynomial time algorithms, it is possible there might be a random self-reduction that is faster than computing the function, but not polynomial time. The notion of random self-reducibility is also related to private result checking because it is possible to privately compute random self-reducible functions. On the other hand, it would be interesting to find new classes of functions which are random self-reducible. We have seen that a large class of functions that have the linearity property or compute polynomial functions are random self-reducible. Is it possible to determine whether or not sorting is random self-reducible?

Some inroads have been made in the study of which functions do not have polynomial time result checkers. Yao [66] has shown that there are functions in $DSPACE(2^{n^{\log n}})$ that do not have polynomial time result checkers. Beigel and Feigenbaum [11], later improved upon this to show that there is a function in $DSPACE(n^{\log^* n})$ that has no polynomial time result checker and is not polynomial time random self-reducible. They also show that if $NEEEXPTIME$ is not a subset of $BPEEEXPTIME$ ($EEEXPTIME$ is the union, over all polynomials p, of $DTIME(2^{2^{2^{p(n)}}})$), then there is a set in NP that has no polynomial time result checker and is not polynomial time random self-reducible. Since there is a complete problem in $EXPTIME$ that has a polynomial time checker [8, Babai Fortnow Lund], there is no correlation between the complexity of computing a function and the complexity of result checking the function. It has not been determined whether or not it is possible to have a polynomial time result checker for complete problems in many other complexity classes; most notably it is not known whether there are polynomial time result checkers for NP-complete problems. [1]

Result checkers, self-testers/correctors have been written for many types of problems: numerical, graph theoretic, algebraic. However, since this approach to program correctness is relatively new, general techniques must be developed in order to write result checkers, self-testers/correctors for new problems, and to improve the existing ones.

We mention some areas that deserve special attention: One new area is that of data-structure checking as introduced in [16, Blum Evans Gemmell Kannan Naor]. They study result checkers for database problems, and for programs that manipulate simple data structures such as stacks. A second important area is that of problems dealing with real numbers. Because computers have finite precision, much of the problem with such programs is to determine what their specifications should be. This is not an issue that is addressed by program result checking, however many interesting problems remain once the specifications have been determined. It would be interesting to see if any of the existing techniques in Chapter 4 can be applied to such problems and if new techniques can be developed. A third area is that of cryptographic protocols. Some initial results about result checking/self-testing/correcting with respect to cryptographic multi-party protocols are given in [51, Micali Rubinfeld].

[1] If one shows a polynomial time result checker for any NP-complete problem, a polynomial time result checker for any other NP-complete problem can be constructed using Beigel's theorem [15, Blum].

Bibliography

[1] Abadi, M., Feigenbaum, J., Kilian, J., "On Hiding Information from an Oracle", *Journal of Computer and System Sciences*, Vol. 39, No. 1, August 1989, pp. 29-50.

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

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

[4] Ajtai, M., personal communication through M. Naor.

[5] Alon, N., Babai, L., Itai, "A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem", *J. Algorithms*, vol. 7, 1986, pp. 567-583.

[6] Babai, L., "Trading Group Theory for Randomness", *Proc. 17th ACM Symposium on Theory of Computing*, 1985, pp. 421-429.

[7] Babai, L., "E-mail and the Power of Interaction", *Proc.* 5*th Structure in Complexity Theory Conference*, 1990.

[8] Babai, L., Fortnow, L., Lund, C., "Non-Deterministic Exponential Time has Two-Prover Interactive Protocols", Technical Report 90-03, University of Chicago, Dept. of Computer Science. Also to appear in *Proceedings of the 31st Annual Symposium on Foundations of Computer Science,*, 1990.

[9] Beaver, D., Feigenbaum, J., "Hiding Instance in Multioracle Queries", *STACS 1990.*

[10] Beaver, D., Feigenbaum, J., Kilian, J., Rogaway, P., "Cryptographic Applications of Locally Random Reductions", AT&T Bell Laboratories Technical Memorandum, November 1989.

[11] Beigel, R., Feigenbaum, J., "On the Complexity of Coherent Sets", AT&T Bell Laboratories Technical Memorandum, February 1990.

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

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

[14] Beame, P., Hastad, J., "Optimal Bounds for Decision Problems on the CRCW PRAM", *Proc. 19th ACM Symposium on Theory of Computing*, 1987.

[15] Blum, M., "Designing programs to check their work", Submitted to *CACM.*

[16] Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M., "Checking the Correctness of Data Storage and Retrieval Algorithms" (tentative title), in preparation.

[17] Blum, M., Kannan, S., "Program correctness checking ... and the design of programs that check their work", *Proc. 21st ACM Symposium on Theory of Computing*, 1989.

[18] Blum, M., Kannan, S., Rubinfeld, R., "A catalogue of checkable problems", in preparation.

[19] Blum, M., Luby, M., Rubinfeld, R., "Program Result Checking Against Adaptive Programs and in Cryptographic Settings", *DIMACS Workshop on Distributed Computing and Cryptography*, 1989.

[20] Blum, M., Luby, M., Rubinfeld, R., "Self-Testing/Correcting with Applications to Numerical Problems," ICSI Technical Report No. TR-90-041.

[21] Blum, M., Luby, M., Rubinfeld, R., "Self-Testing/Correcting with Applications to Numerical Problems," *Proc. 22th ACM Symposium on Theory of Computing*, 1990.

[22] Blum, M., and Micali, S., "How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits", *SIAM J. on Computing*, Vol. 13, 1984, pp. 850-864.

[23] Blum, M., and Raghavan, P., "Program Correctness: Can One Test for It?", IBM T.J. Watson Research Center Technical Report (1988).

[24] Chandra, A., Fortune, S., Lipton, R., "Unbounded Fan-in Circuits and Associative Functions", *Proc. 15th ACM Symposium on Theory of Computing*, 1983.

[25] 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.

[26] Coppersmith, D., Winograd, S., "Matrix Multiplication via Arithmetic Progressions", *Proc. 19th ACM Symposium on Theory of Computing, 1987.*

[27] De Millo, R.A., Lipton, R.J., Perlis, A.J., "Social Processes and Proofs of Theorems and Programs," *Communications of the ACM*, May 1979, Vol. 22, No. 5, pp. 271-280.

[28] Feigenbaum, J., Kannan, S., Nisan, N., "Lower Bounds on Random Self-Reducibility", *Proc. 5th Structure in Complexity Theory Conference*, 1990.

[29] Feldman, P., "The Optimum Prover Lies in PSPACE", manuscript, 1986.

[30] Fiat, A., Naor, M., Personal communication through Moni Naor.

[31] Fich, F., "New bounds for parallel prefix circuits", *Proc. 15th ACM Symposium on Theory of Computing*, 1983, pp.27-36.

[32] Fortnow, L., "Complexity-Theoretic Aspects of Interactive Proof Systems", Tech Report MIT/LCS/TR-447, May 1989.

[33] Fortnow, L., Rompel, J., Sipser, M., "On the Power of Multi-Prover Interactive Protocols", Proc. 3^{rd} Structure in Complexity Theory Conference, 1988, pp. 156-161.

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

[35] Furst, M., Saxe, J., Sipser, M., "Parity, Circuits and the Polynomial Time Hierarchy", *Math. Systems Theory*, vol. 17, 1984, pp.13-28.

[36] Goldberg, M., Spencer, T., "A New Parallel Algorithm for the Maximal Independent Set Problem," *Proceedings of the 28th Annual Symposium on Foundations of Computer Science, 1987.*

[37] Goldwasser, S., Micali, S., Rackoff, C., The Knowledge Complexity of Interactive Proof Systems", *SIAM J. Comput.*, 18(1),1989, pp. 186-208.

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

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

[40] Karloff, H., "A Las Vegas RNC Algorithm for Maximum Matching," *Combinatorica*, vol. 6, 1986, pp.387-392.

[41] 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.

[42] Karp, R., Ramachandran, V., "A Survey of Parallel Algorithms for Shared-Memory Machines", UC Berkeley Technical Report No. UCB/CSD 88/408.

[43] Karp, R., Upfal, E., Wigderson, A.,"Constructing a perfect matching is in random NC", *Combinatorica*, vol. 6, 1986, pp.35-48.

[44] Karp, R., Upfal, E., Wigderson, A.,"The Complexity of Parallel Search", *J. Comp. Syst. Sci.*, 1988.

[45] Kompella, K., "Efficient Checkers for Cryptography", manuscript.

[46] Kompella, K., Adleman, L.,"Checkers for RSA", manuscript.

[47] Ladner, R., Fischer, M.,"Parallel Prefix Computation", JACM, vol. 27, 1980, pp.831-838.

[48] Lipton, R., "New directions in testing", manuscript.

[49] Luby, M., "A Simple Parallel Algorithm for the Maximal Independent Set Problem", SIAM J. Comput., vol. 15, 1986, pp. 1036-1053.

[50] Lund, C., Fortnow, L., Karloff, H., Nisan, N., "Algebraic Methods for Interactive Proof Systems", to appear in *Proceedings of the 31st Annual Symposium on Foundations of Computer Science,*, 1990.

[51] Micali, S., Rubinfeld, R., "Privacy Implies Correctness", manuscript.

[52] Mulmuley, K., Vazirani, U., Vazirani, V., "Matching is as Easy as Matrix Inversion", *Proc. 19th ACM Symposium on Theory of Computing*, 1987.

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

[54] Randall, D., "Efficient Random Generation of Invertible Matrices", manuscript.

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

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

[57] Rubinfeld, R., "Designing Checkers for Programs that Run in Parallel", ICSI Technical Report No. TR-90-040.

[58] Rubinfeld, R. "Batch Checking for the Mod Function", manuscript, 1990.

[59] Rubinfeld, R., Sudan, M., "Self-Testing Polynomial Functions and Approximate Functions", in preparation.

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

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

[62] Shamir, Adi, "IP=PSPACE", to appear in *Proceedings of the 31st Annual Symposium on Foundations of Computer Science, 1990.*

[63] Strassen, V., “Gaussian Elimination is not Optimal”, *Numerische Mathematik*, 13, 1969, pp. 354-356.

[64] Van Der Waerden, B.L., *Algebra*, Vol. 1, Frederick Ungar Publishing Co., Inc., pp. 86-91, 1970.

[65] Wilkes, M., *Memoirs of a Computer Pioneer*, The MIT Press, Cambridge, Mass., p. 145, 1985.

[66] Yao, A., “Coherent Functions and Program Checking”, *Proc. 22th ACM Symposium on Theory of Computing*, 1990.