Program Result Checking Against Adaptive Programs and in Cryptographic Settings (Extended Abstract)

Manuel Blum [*]
Computer Science Division
U.C. Berkeley
Berkeley, California 94720

Michael Luby
International Computer Science Institute
Berkeley, California 94704

Ronitt Rubinfeld [†]
Computer Science Division
U.C. Berkeley
Berkeley, California 94720

May 17, 1990

Abstract

The theory of program result checking introduced in [Blum] allows one to check that a program P correctly computes the function f on input x. The checker may use P's outputs on other inputs to help it check that $P(x) = f(x)$. In this setting, P is always assumed to be a fixed program, whose output on input x is a function $P(x)$. 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 a checker that works for such a program an *adaptive checker*. We consider the case where there is an adaptive program that supposedly computes f running on each of several noninteracting machines. We design adaptive checkers that work for a constant number of independent and noninteracting programs.

We also consider the following cryptographic scenario: A user wants to evaluate function f on input x using program P running on another machine. As in 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) as in [Abadi Feigenbaum Kilian] [Beaver Feigenbaum]. We call a program that satisfies the above constraints a *private checker*. As is the case for adaptive checking, we consider the case where there is a program that supposedly computes f on each of several noninteracting machines. We design private checkers that work for a constant number of independent and noninteracting programs.

The adaptive and private checkers given are general techniques that work for a variety of numerical problems, including integer multiplication, modular multiplication, matrix multiplication, the mod function, integer division, modular exponentiation and polynomial multiplication.

[*]Supported by NSF Grant No. CCR 88-13632.
[†]Supported by an IBM Graduate Fellowship and NSF Grant No. CCR 88-13632.

1 Introduction

Consider the task of writing a program P to evaluate a function f. One of the main difficulties of this task is that when P is implemented, it is difficult to verify that $P(x) = f(x)$ for all inputs x. The theory of program result checking, introduced by [Blum], provides a way of doing this. Intuitively, a checker C for f makes calls to a program P that supposedly computes f. C is a program that is much "different" than any correct program for computing f. On input x and β, C outputs $P(x)$ and either "FAULTY" or "PASS". C always outputs "PASS" if P is a correct program for f *on all inputs*, whereas if $P(x) \neq f(x)$ then C outputs "FAULTY" with probability $\geq 1 - \beta$.

We want to ensure that the checker is in some way different than the program being checked, so that bugs in the checker are not the same as bugs in the program. This can be done in several ways. The way which we use here is to require that the running time of the checker, not including the time required by calls to the program, be asymptotically faster than the running time of any known correct program which computes $f(x)$. We call a checker that has this property *different*.

Since the checker is run each time that the program is used, it is also important that the total running time, including calls to the program, be efficient. We say a checker is *efficient* if the total running time is a constant multiplicative factor times the running time of the program. We say a checker is $f(n)$-*efficient* if the total running time is $f(n)$ multiplied by the running time of the program.

The definition of a checker assumes that the program P is static, i.e. it computes some function, though not necessarily the function f (P can be a probabilistic program in this definition, as long as the output of P on input x only depends on x, and not other inputs). 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 call such a program that can modify itself and its subsequent computation an *adaptive program.* In this model, we assume that the program is an adversary that can give answers depending on all the previous questions asked of it by the checker. This is a restriction of the model used in interactive proof systems, in which the role of the verifier is played by the 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 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 multi-prover interactive proof systems where k is the number of provers.

Definition 1: Program C is an *adaptive checker* for f if C is a checker for f that also works with respect to adaptive programs that supposedly compute f.

Definition 2: Program C is a k-*adaptive checker* if C is a checker for k adaptive programs which do not communicate among themselves.

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

Many checkers that have been found are also adaptive checkers. For example, it can easily be seen that the GCD checker in [Adleman Huang Kompella] and that all of the checkers given in [Blum Kannan] are adaptive. Other checkers do not work for an adaptive program. Examples of such checkers are the ones in [Blum Luby Rubinfeld], where adaptive programs can easily fool the checkers. At the present time we see no way to convert such a checker into an adaptive checker. However, if more than one copy of the program exists, we show that a checkers based on the methods in [Blum Luby Rubinfeld] 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 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 [Abadi Feigenbaum Kilian] and later extended in [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 [Abadi Feigenbaum Kilian] [Beaver Feigenbaum] where the checker may only ask the program questions of the form "What is the value of $f(x)$?".

We introduce some notation in order to define a private 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 [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 checker to the programs on input x. An example is $L(x) = |x|$, i.e. the 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 3: 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 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 checker to output "FAULTY" on input x if $P(x) \neq f(x)$ is insufficient. We define a private checker as follows:

Definition 4: A program C is a (k, L)-*private 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) $Pr[C(x) = f(x) \text{ or } C(x) = \text{"}FAULTY\text{"}] \geq 1 - \beta$.

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 5: A program C is a (k, L)-*private/adaptive checker* if it is a k-adaptive checker and a (k, L)-private checker.

We ask that the adaptive and private checkers be *different*, and as efficient as possible. A (k, L)-

private checker or a k-adaptive checker is $f(n)$-efficient if the total work done *by all k programs* and the checker is $f(n)$ multiplied by the running time of the program.

2 Description of Results and Related Work

We present general techniques for constructing simple to program and efficient (k, L)-private and k-adaptive checkers, for a constant k 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 checkers given in this paper are all based on the algorithms given in [Blum Luby Rubinfeld], 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 checker algorithms are both efficient and different. Furthermore, the 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.

[Abadi Feigenbaum Kilian] show that there is not likely to be a $(1, |x|)$-private 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.

The notion of random-self-reducibility, as defined in [Feigenbaum Kannan Nisan], is related to private checking because it is possible to privately compute random-self-reducible functions. Recently [Feigenbaum Kannan Nisan] have shown that random boolean functions are not k-random-self-reducible for any polynomial k, and that if a function is 2-random-self-reducible, then the function can be computed nonuniformly in nondeterministic polynomial time.

[Beaver Feigenbaum] [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 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 [Fortnow Rompel Sipser], there is a general technique for turning any checker into a 2-adaptive checker. This technique can actually be used for many of the 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 checking techniques in [Blum Luby Rubinfeld], which makes $O(\log n)$ calls to the program, into an adaptive checker which is of the same efficiency as the original checker. The techniques of [Fortnow Rompel Sipser] yield an adaptive checker that is slower than the original checker by a multiplicative factor of $O(\log n)$.

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

3 Three Properties of Functions

The three properties described in this section are also described in [Blum Luby Rubinfeld] and for completeness we include them here. We build checkers for functions that have these properties. We include the running example of multiplication in order to show how the definitions are applied to a specific problem.

Let $\hat{U}$ denote the domain of f and let $\hat{x} \in \hat{U}$ be an input to f. Let $\hat{I}^1, \hat{I}^2, \ldots$ be a sequence of subsets of $\hat{U}$ such that $\hat{U} = \cup_{n \in \mathcal{N}} \hat{I}^n$. The superscript n indicates the *size* of the problem. Let $\mathcal{D} = \{D_n | n \in \mathcal{N}\}$ be a collection of probability distributions such that D_n is a distribution on $\hat{I}^n$. Let P be a program that supposedly computes f.

In the case of integer multiplication, $\hat{U} = \mathcal{N} \times \mathcal{N}$ is the set of pairs of natural numbers and $\hat{x} = (x_1, x_2) \in \hat{U}$ where $f(\hat{x}) = x_1 \cdot x_2$. $\hat{I}^n$ is the set of pairs of numbers of length at most n ($\{x | 0 \leq x \leq 2^n\} \times \{x | 0 \leq x \leq 2^n\}$).

3.1 Computability by Random Inputs

The first property that we are interested in is that any particular instance of the problem can be expressed as the solution to a few random instances of the same size:

Let $\hat{x} \in \hat{I}^n$ be an instance of the problem. Let $c_1(n)$ be an easily computable function of n (in the following we use c_1 in place of $c_1(n)$ for brevity and because in most cases $c_1(n)$ is a constant). Informally, the property is that $f(\hat{x})$ can be expressed as an easily computable function of $f(\hat{a}_1), \ldots, f(\hat{a}_{c_1})$, where $\hat{a}_1, \ldots, \hat{a}_{c_1}$ are each randomly distributed in $\hat{I}^n$ according to D_n.[1] More formally, there is a generation procedure G and a computation F, both easily computable, that interact as follows. For all n and for all $\hat{x} \in \hat{I}^n$:

(1) $G(n, \hat{x}, R) = \hat{a}_1, \ldots, \hat{a}_{c_1}, A$, where R is the bit string used as a random source of bits. When R is randomly and uniformly distributed then, for each $k = 1, \ldots, c_1$, $\hat{a}_k$ is randomly distributed in $\hat{I}^n$ according to D_n. A is auxiliary information generated in the process which is used in part (2).

(2) $F(n, \hat{x}, G(n, \hat{x}, R), \alpha_1, \ldots, \alpha_{c_1})$ evaluates to a possible output of an input in $\hat{I}^n$. Intuitively, $\alpha_1, \ldots, \alpha_{c_1}$ are the answers that the program being checked produces on inputs $\hat{a}_1, \ldots, \hat{a}_{c_1}$, respectively. We require that:

If, for all $k = 1, \ldots, c_1$, $\alpha_k = f(\hat{a}_k)$ then $F(n, \hat{x}, G(n, \hat{x}, R), \alpha_1, \ldots, \alpha_{c_1}) = f(\hat{x})$.

For integer multiplication, the conditions are fulfilled as follows.

[1] However, no independence between these random variables is needed, e.g. given the value of $\hat{a}_1$ it is not necessary that $\hat{a}_2$ be randomly distributed in $\hat{I}^n$ according to D_n.

(1) *First, generate two truly random and independent bit strings r_1 and r_2, each of length n, using a random bit string R of length $2n$. Then, compute δ_1 and δ_2, where $\delta_1 = 1$ if $r_1 > x_1$ and $\delta_1 = 0$ otherwise and δ_2 is calculated analogously with respect to r_2 and x_2. Let $s_1 = \delta_1 2^n + x - r_1$ and $s_2 = \delta_2 2^n + x_2 - r_2$. It can be easily verified that each of the pairs $\hat{a}_1 = (r_1, r_2)$, $\hat{a}_2 = (r_1, s_2)$, $\hat{a}_3 = (s_1, r_2)$ and $\hat{a}_4 = (s_1, s_2)$ are randomly and uniformly distributed in $\{0,1\}^n \times \{0,1\}^n$. Then, $G(i, \hat{x}, R) = \hat{a}_1, \hat{a}_2, \hat{a}_3, \hat{a}_4, I$ where $I = (r_1, r_2, s_1, s_2, \delta_1, \delta_2)$.*

(2) *Define $F(n, \hat{x}, G(n, \hat{x}, R), \alpha_1, \ldots, \alpha_4) = \alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 - (r_1 + s_1)\delta_2 2^n - (r_2 + s_2)\delta_1 2^n + \delta_1 \delta_2 2^{2n}$. It is easy to verify that F satisfies the requirements stated above.*

The generation and computation steps require the checker to perform a constant number of shifts, comparisons and additions on numbers of length n.

3.2 Computability by Smaller Inputs

Informally, the property is that there is a constant c_2 such that for all $\hat{x} \in \hat{I}^n$, $f(\hat{x})$ can be expressed as an easily computable function of $f(\hat{a}_1), \ldots, f(\hat{a}_{c_2})$, where $\hat{a}_1, \ldots, \hat{a}_{c_2}$ are each in $\hat{I}^{n-1}$. More formally, there is a generation procedure G' and a computation F', both easily computable, that interact as follows. For all $n > 1$ and for all $\hat{x} \in \hat{I}^n$:

(1) $G'(n, \hat{x}) = \hat{a}_1, \ldots, \hat{a}_{c_2}, I$, where, for each $k = 1, \ldots, c_2$, $\hat{a}_k \in \hat{I}^{n-1}$. I is auxiliary information generated in the process which is used in part (2).

(2) $F'(n, \hat{x}, G'(n, \hat{x}), \alpha_1, \ldots, \alpha_{c_2})$ evaluates to a possible output of an input in $\hat{I}^n$. Intuitively, $\alpha_1, \ldots, \alpha_{c_2}, \beta$ are the answers that the program being checked produces on inputs $\hat{a}_1, \ldots, \hat{a}_{c_2}, \hat{x}$, respectively. We require that:

If, for all $k = 1, \ldots, c_2$, $\alpha_k = f(\hat{a}_k)$ then $F'(n, \hat{x}, G'(n, \hat{x}), \alpha_1, \ldots, \alpha_{c_2}) = f(\hat{x})$.

For example, for integer multiplication, this condition is fulfilled as follows. In fact, instead of reducing to inputs one bit smaller we reduce to half the number of bits. Assume n is even.

(1) Let x_1^L be the most significant $n/2$ bits of x_1 and let x_1^R be the least significant $n/2$ bits of x_1. Define x_2^L and x_2^R analogously with respect to x_2. Let $\hat{a}_1 = (x_1^R, x_2^R)$, $\hat{a}_2 = (x_1^L, x_2^R)$, $\hat{a}_3 = (x_1^R, x_2^L)$ and $\hat{a}_4 = (x_1^L, x_2^L)$. Then, $G'(n, \hat{x}) = \hat{a}_1, \hat{a}_2, \hat{a}_3, \hat{a}_4$.

(2) Define $F'(n, \hat{x}, G'(n, \hat{x}), \alpha_1, \ldots, \alpha_4) = \alpha_1 + (\alpha_2 + \alpha_3) \cdot 2^{n/2} + \alpha_4 \cdot 2^n$. It is easy to verify that F' satisfies the requirements stated above.

3.3 Computability by Random Homomorphisms

Let G be a finite group with group operation $\circ$ and with generators $g_1, \ldots, g_{c_3}$ 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 relatively hard to compute compared to either $\circ$ or $\circ'$.

Let $\mathcal{U}_G$ be the uniform probability distribution on G. We say that f has the *computability by random homomorphisms* property if:

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

(2) F is an easily computable function with the property that, for any pair $x_1, x_2 \in G$, $F(x_1, x_2) \in G'$ and furthermore $f(x_1 \circ x_2) = f(x_1) \circ' f(x_2) \circ' F(x_1, x_2)$. We call this property *linear consistency.* In all of our applications except for integer multiplication, $F(x_1, x_2) = 0'$ 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 is an easily computable function with the property that, for any $z \in G$, $F_i(z) \in G'$ and furthermore $f(z \circ g_i) = f(z) \circ' F_i(z)$. 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 computability by random homomorphisms property is a special case of the computability by random inputs property. The name comes from the fact that f is a group homomorphism in all of our applications except for integer multiplication, in which case f exhibits properties similar to a group homomorphism.

4 Main Theorems

We first show how to construct a checker that is both adaptive *and* private for any function computable by random inputs and computable by smaller inputs. The checker is an adaptation of the self-testing/correcting pair using the method based on reduction to smaller sized inputs given in [Blum Luby Rubinfeld].

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 "FAULTY" then the entire checker program outputs "FAULTY" and halts immediately.

Program $Check[l, \hat{x}]$**:** The inputs are $l \in \mathcal{N}$, $\hat{x} \in \hat{I}^l$.

```
    For $i = 1, \ldots, l$, call $Random\_test[i]$.
    $answer = Random\_Compute[l, \hat{x}]$
    Output ($answer$,"PASS")
```

Subroutine $Random\_Test[n]$**:**

```
    Choose $\hat{x}$ at random from $\hat{I}^n$.
    If $n = 1$ then compute $f(\hat{x})$ directly
    For $1 \leq i \leq c_1, y_i \leftarrow P_i(\hat{x})$
    If there is an $i$ s.t. $f(\hat{x}) \neq y_i$ then output "FAULTY"
  Elseif $n > 1$ then:
    Use computability by smaller inputs to generate $G'(i, \hat{x}) = \hat{a}_1, \ldots, \hat{a}_{c_2}, I$.
    For all $k = 1, \ldots, c_2$ let $y_k \leftarrow Random\_compute[n-1, \hat{a}_k]$.
    For $1 \leq i \leq c_1, z_i \leftarrow P_i(\hat{x})$
    If there is an $i$ such that $F'(n, \hat{x}, G'(n, \hat{x}), y_1, \ldots, y_{c_2}) \neq z_i$ then output "FAULTY"
```

Subroutine *Random_Compute*$(n, \hat{x})$**:**

```
    Use computability by random inputs to generate $G(n, \hat{x}, R) = \hat{a}_1, \ldots, \hat{a}_{c_1}, I$,
      where $R$ is a random, independent and uniformly distributed bit string.
    For $i = 1, \ldots, c_1$, $\alpha_i \leftarrow P_i(\hat{a}_i)$.
    $answer \leftarrow F(n, \hat{x}, G(n, \hat{x}, R), \alpha_1, \ldots, \alpha_{c_1})$
```

This checker is based on the checker in [Blum Luby Rubinfeld] 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 in [Blum Luby Rubinfeld] do not work in this setting. In fact, a naive implementation of the protocol described in [Blum Luby Rubinfeld] can be fooled by c_1 adaptive programs, because the adaptive program can figure out where the 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 [Fortnow Rompel Sipser], one can simply transform the checker in [Blum Luby Rubinfeld] into a 2-adaptive checker, with an additional cost of $O(\log n)$ multiplicative overhead in the running time over the original checker. On the other hand, the adaptive checker presented here is as efficient as the original checker of [Blum Luby Rubinfeld].

Theorem 1: 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 checker, where $T(x) = L(x)$ is the *size* of x.

Proof of Theorem 1: The intuition behind why this 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 the first line of Random-Test or whether the question was generated at random within Random-Compute. We show that this is enough to get a c_1-adaptive 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 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 $\hat{I}^r$ for each $r = 1, \ldots, l$.) Some of the questions are generated in Random-Test, and some of the questions are generated in Random-Compute in order to verify the questions in Random-Test. Notice that the questions asked in Random-Test 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 Random-Test rather than Random-Compute. This is because Random-Test only makes one call to P_i on inputs of size r, whereas

Random-Compute 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 Random-Test, 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 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 "FAULTY". ■

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

Corollary 1.1: There is a $(4, L)$-private/adaptive checker for integer multiplication, where $L(x) = |x|$. The checker is both different and efficient.

Corollary 1.2: There is a $(2, L)$-private/adaptive checker for integer division $f(a, b) = (a \, div \, b, a \bmod b)$, where $L(a, b) = \{|a|, b\}$. The checker is both different and $(\log n)$-efficient.

Corollary 1.3: There is a $(4, L)$-private/adaptive checker for modular exponentiation $f(a, x, q) = a^x \bmod q$, where $L(a, x, q) = \{|a|, |x|, q\}$. The checker is both different and $(\log^2 n)$-efficient.

Corollary 1.4: There is a $(4, L)$-private/adaptive checker for matrix multiplication over a finite field $f(A, B) = A \cdot B$, where $L(A, B) = \{|A|, |B|\}$. The checker is both different and efficient.

Corollary 1.4: There is a $(4, L)$-private/adaptive checker for polynomial multiplication over a finite field $f(a(x), b(x)) = a(x) \times b(x)$, where $L(a, b) = \{degree(a), degree(b)\}$. The checker is both different and efficient.

It is also possible to construct an adaptive and private checker for any function computable by random homomorphisms. The checker is an adaptation of the self-testing/correcting pair using the method based on computability by random homomorphisms given in [Blum Luby Rubinfeld].

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

Proof idea of Theorem 2: [Fortnow Rompel Sipser] show how to transform any checker into a 2-adaptive checker by simply running the original checking protocol with the first program. If the original 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 checker returns "PASS". Otherwise, if the original checker would have returned "FAULTY", or if the second program answers differently than the first, the adaptive checker returns "FAULTY". An adaptation of the algorithm for self-testing/correcting based on computability by random homomorphisms in [Blum Luby Rubinfeld] combined with the technique of [Fortnow Rompel Sipser] proves Theorem 2. ■

Corollary 2.1: There is a 2-adaptive and $(5, L)$-private/adaptive checker for modular multiplication $f(a, b, q) = a \times b \bmod q$, where $L(a, b, q) = \{|a|, |b|, q\}$. The checkers are both different and efficient.

Corollary 2.2: There is a 2-adaptive and $(4, L)$-private/adaptive checker for the mod function $f(a, q) = a \bmod q$, where $L(a, q) = \{|a|, q\}$. The checkers are both different and efficient.

Corollary 2.3: There is a 2-adaptive and $(4, L)$-private/adaptive checker for the exponentiation mod a prime function $f(a, x, p) = a^x \bmod p$, where $L(a, x, p) = \{a, |x|, p\}$. The checkers are both different and efficient.

5 Open Questions

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

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

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

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

6 Acknowledgements

We thank for Oded Goldreich, Sampath Kannan, Joan Feigenbaum, Silvio Micali and Shafi Goldwasser for many enlightening discussions and for technical suggestions.

References

[Adleman Huang Kompella] Adleman, L., Huang, M., Kompella, K., Personal communication through L. Adleman.

[Abadi Feigenbaum Kilian] 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.

[Babai Fortnow Lund] 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.

[Beaver Feigenbaum] Beaver, D., Feigenbaum, J., “Hiding Instance in Multioracle Queries”, To appear in Proceedings of STACS 1990.

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

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

[Blum Kannan] Blum, M., Kannan, S., "Program correctness checking ... and the design of programs that check their work", *STOC 1989.*

[Blum Luby Rubinfeld] Blum, M., Luby, M., Rubinfeld, R., "Program Result Checking Against Adaptive Programs and in Cryptographic Settings", to appear in proceedings of *STOC 1990.*

[Feigenbaum Kannan Nisan] Feigenbaum, J., Kannan, S., Nisan, N., "Lower Bounds on Random-Self-Reducibility", to appear in proceedings of Structure in Complexity Theory Conference, 1990.

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

[Fortnow Rompel Sipser] 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.

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

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

[Lund Fortnow Karloff Nisan] Lund, C., Fortnow, L., Karloff, H., Nisan, N., "The Polynomial Time Hierarchy has Interactive Proofs", Manuscript.

[Lipton] Lipton, R., "New directions in testing", Manuscript.

[Rubinfeld] Rubinfeld, R. "Designing checkers for programs that run in parallel", Manuscript.

[Shamir] Shamir, A., "IP=PSPACE", Manuscript.