Checking the Correctness of Memories*

Manuel Blum† Will Evans† Peter Gemmell† Sampath Kannan‡
Moni Naor§

September 7, 1995

Abstract

We extend the notion of program checking to include programs which alter their environment. In particular, we consider programs which store and retrieve data from memory. The model we consider allows the checker a small amount of reliable memory. The checker is presented with a sequence of requests (on-line) to a data structure which must reside in a large but unreliable memory. We view the data structure as being controlled by an adversary. We want the checker to perform each operation in the input sequence using its reliable memory and the unreliable data structure so that any error in the operation of the structure will be detected by the checker with high probability.

We present checkers for various data structures. We prove lower bounds of $\log n$ on the amount of reliable memory needed by these checkers where n is the size of the structure. The lower bounds are information theoretic and apply under various assumptions. We also show time-space tradeoffs for checking random access memories as a generalization of those for coherent functions.

1 Introduction

The program checking model was introduced in [3] and several subsequent papers [4, 1, 11] have provided checkers for classical computational problems. The model was introduced as a practical means of checking that programs for these problems are correct. Rather than certifying that the program is always correct, a program checker certifies that *on any given input* the program is correct. Of course, an incorrect program may be correct on some inputs, and for those inputs the checker may or may not detect an error. The requirement is that the checker detect incorrect behavior on the given input.

In the context of program checking, programs have been thought of merely as computing a function and not as having any side effects. An important problem is extending the concept

*Supported in part by NSF grant CCR 88-13632. A preliminary version of this paper appeared in the 32nd IEEE Symposium on Foundations of Computer Science.

†Department of Computer Science, University of California at Berkeley, CA 94720

‡DIMACS, Rutgers University, P.O. Box 1179, Piscataway, NJ 08855

§IBM Almaden, 650 Harry Road, San Jose, CA 95120

of program checking to computations that cannot be modeled this way. Such an extended notion is required to check programs that implement storage and retrieval from memory. Further, checking the correctness of these programs is of great practical importance. In this paper, we define a suitable model for checking such programs and present checkers for various problems of storage and retrieval.

As in the program checking case, we only require that the checker certify the correct operation of the program on any given input. However, when checking storage and retrieval, the input does not result in a single output. Instead, the input is a sequence of operations, such as reads and writes to memory, and each operation may have an associated output. In addition, the output of an operation in the input sequence will usually depend on the preceding operations in the sequence. Thus checkers for programs which interact with memory must check that the output not only follows the problem specification but also is consistent with the input sequence.

The question of checking a sequence of stores and retrieves from a random access memory has been addressed by the papers of Goldreich [7] and Ostrovsky [15]. These two papers actually solve the harder problem of software protection against a very powerful adversary. Consequently, the overheads involved in checking the sequence of memory accesses is quite large. In this paper, we provide checkers not only for RAMs but also for the more restricted problems of stores and retrieves from stacks and queues with much smaller overhead.

A more detailed description of the contents of this paper will be given after the model is discussed.

2 Model

A data structure is defined by specifying the output of each data structure operation in any sequence of operations performed on the data structure starting in some initial configuration. For example the definition of a stack assigns to the sequence "push a, push b, pop, pop" the outputs "$\emptyset$, $\emptyset$, b, a" where $\emptyset$ indicates no output.

We think of the data structure as residing in a large unreliable memory. In fact, we view the data structure as being controlled by an adversary. The user interacts with the data structure by presenting it with a sequence of operations. The checker's job is to detect any error in the behavior of the data structure while performing the user's operations. The checker is allowed only a small amount of reliable memory in order to achieve this goal.

An error occurs if any value returned from the data structure does not match the corresponding value entered into the data structure. For example, in a queue if v is the i^{th} value enqueued and w is the i^{th} value dequeued, then $v \neq w$ is an error. In the case of the RAM, a read at address i must return the last value written to address i. Note that entering a value into a data structure does not produce an output and thus has no formally-acceptable notion of being incorrectly executed.

Definition: A *memory checker* for a data structure $\mathcal{D}$ is a probabilistic Turing machine C with five tapes: a read only input tape from which the checker reads user specified operations to $\mathcal{D}$, a write only output tape on which the checker writes the output of each operation or declares the implementation of $\mathcal{D}$ to be BUGGY, a read/write worktape called the *reliable*

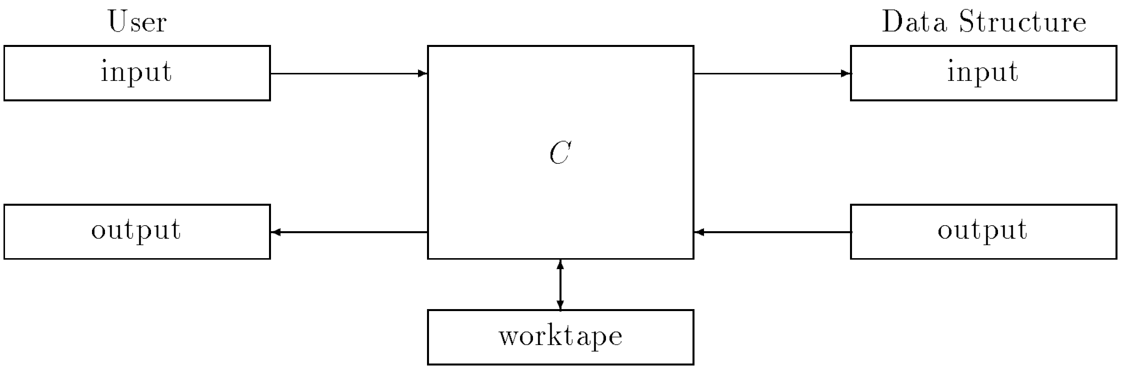

Figure 1: Memory Checker interaction with User and Data Structure

or *private* checker memory, a write only input tape on which the checker specifies operations to $\mathcal{D}$, and a read only output tape from which the checker reads the output of each operation (as determined by some implementation of $\mathcal{D}$).

The checker C is presented with operations to $\mathcal{D}$ on its input tape. It is required to write the output of each operation (or BUGGY) on its output tape before the next operation is presented.

Finally, for all implementations D of data structure $\mathcal{D}$ and for all user operation sequences of length polynomial in n (the size of the data structure):

- If D's output is correct for all operations in the sequence then C's output is correct with probability $> 3/4$.
- If D's output is incorrect for some operation then C outputs BUGGY with probability $> 3/4$.

See Figure 1 for a pictorial depiction of the memory checker.

If C's reliable memory is sufficiently large then C can check the operation of D simply by performing the data structure operations itself using its worktape to hold the data structure and checking that D always agrees. We will be interested in obtaining checkers which have small worktapes; typically size logarithmic in the size of the data structure. Note that by restricting the size of the checker's worktape we force the checker to be different from the data structure implementation.

The definition of a memory checker does not specify when the checker should output BUGGY if it detects an error. Ideally, we would like the checker to output BUGGY immediately after an errant operation. We call this type of checker an *on-line* checker. Alternatively, if we allow the checker to wait until the end of the sequence of operations to output BUGGY then the checker is an *off-line* checker. In either case the checker is required to output a result for each operation before the user presents the next operation.

In addition, we distinguish checkers on one other score: Checkers which use the data structure to store information other than what the user requests are called *invasive*. Checkers which do not introduce their own operations are called *noninvasive*. For example, when the user issues a write request of some value, an invasive checker may write a time stamp as well as the value into memory. Or an invasive checker may write an encrypted version of the value. A noninvasive checker must write only the user's value in the memory.

Finally, we would like to design checkers which introduce very little overhead per user operation. Ideally, the checker would perform a constant number of data structure operations and a constant amount of additional work per user operation. Some of the checker's that are presented in this paper perform an amount of work per user operation which is proportional to the logarithm of the data structure size.

In the next section, we describe several hashing tools from the literature that are used here to design checkers. Section 4 presents off-line checkers for queues, stacks, and RAMs. Section 5 gives on-line checkers for these data structures.

Finally, in section 6 we prove two lower bounds: The first is an $\Omega(\log n)$ lower bound on the size of the checker's private memory for checking the correctness of an n-bit string stored and retrieved from memory. This lower bound is information theoretic and hence very robust. It holds for all the different types of checkers discussed in this paper, even with cryptographic assumptions, with the checker being given access to a random oracle, and with the checker being allowed to take more than polynomial time per request. On the other hand, it is very simple to design an off-line, noninvasive checker for this problem with $O(\log n)$ bits of memory, running in polynomial time and using no cryptographic assumptions.

The second lower bound has to do with on-line, noninvasive checkers. We show that if the checker has m bits of memory and is allowed at most t accesses to main memory per request, then the size of the main memory cannot be much larger than mt.

3 Hashing tools

Several hashing techniques are used in the paper. Some of them rely on cryptographic assumptions while others do not. We review these hashing techniques in this section.

3.1 ϵ-biased hash functions

This hashing scheme is drawn from [13]. We briefly describe the result in a communication complexity setting. Suppose two players A and B have n-bit strings x and y respectively and would like to decide if $x = y$. The scheme in [13] allows A to define a hash function h using $O(\log n + k)$ random bits such that $h(x)$ is small ($O(k)$ bits) and $h(x) = h(y)$ with probability $\leq 1/2^k$ if $x \neq y$.

The hash function description is treated as a source of $O(\log n + k)$ bits. These bits are expanded into $l = O(k)$ *distinguisher* strings $r_1, r_2, \ldots, r_l$, each of length n, which are "random" enough to ensure that if $x \neq y$ then the inner products, $\langle x, r_i \rangle$ agree with the corresponding inner products, $\langle y, r_i \rangle$ mod 2 for all i with probability $\leq 1/2^k$. A further nice property of this scheme is that for any j, the j^{th} bit of any r_i can be generated using a constant number of $\log n$-bit operations.

In the communication setting this allows B to determine if $x = y$ with high probability using $O(\log n + k)$ bits of communication. In the checking scenario this allows the checker to use $O(\log n + k)$ bits of reliable (and secret) memory to fingerprint an input which is n bits long.

It is interesting to note that hashing mod a random prime of length $O(\log n)$ provides a scheme for testing equality that achieves a constant probability of error, while this scheme produces an inverse polynomial probability of error with the same number of bits. Furthermore, the fact that one can generate any particular bit in constant time allows us to compute this hash function as the string to be hashed is revealed bit by bit.

We now list two (cryptographic) techniques for hashing.

3.2 Pseudorandom functions

A family of *pseudorandom functions*, as defined by Goldreich, Goldwasser and Micali [8], is a collection of functions that has the property that a random member of it is indistinguishable from a random function, yet it has a succinct representation and can be efficiently computed.

More precisely, let $D(n)$ be a sequence of domains such that $|D(n)|$ is polynomial in n, let $k(n) = n^\epsilon$ for some $\epsilon > 0$. and let $\ell(n)$ be bounded by a polynomial in n ($\ell(n)$ is possibly subpolynomial). A family of pseudorandom function consists of a sequence of polynomial time computable functions F_n, that on input $S \in \{0,1\}^{k(n)}$ and $x \in D(n)$ output a member of $\{0,1\}^{\ell(n)}$. We denote by $f_S(x)$ the function $D \mapsto \{0,1\}^\ell$ obtained from $F_n(S,x)$ by fixing S. For F to be pseudo-random means that for any probabilistic polynomial time adversary $\mathcal{A}$ that attempts to distinguish f_S from a truly random function the following holds. $\mathcal{A}$ is given either (a) a random function $D(n) \mapsto \{0,1\}^{\ell(n)}$ or (b) a random $S \in \{0,1\}^{k(n)}$ is chosen. $\mathcal{A}$ can choose adaptively $x \in D(n)$ and in case (a) gets a random value (if it queries twice the same x it gets the same value) and in case (b) it gets $f_S(x)$. After polynomial number of queries $\mathcal{A}$ should guess whether it is case (a) or (b). For all polynomial p for sufficiently large n

$$\mathrm{Prob}[\mathcal{A} \text{ outputs } \text{'a'} \mid \text{case (a)}] - \mathrm{Prob}[\mathcal{A} \text{ outputs } \text{'a'} \mid \text{case (b)}] < \frac{1}{p(n)}$$

Goldreich, Goldwasser and Micali [8]) show how to construct such functions F_n based on any pseudorandom generator, which it turn can be based on any one-way hash function [9, H].

We assume that the reliable (and secret) memory of the checker can store the seed S of pseudorandom function f. The contents of memory cells can be authenticated by adding tags: if we wish to store value v in location i, the checker stores both v and the tag $f_S(i,v)$ in location i. Here $f_S(i,v)$ is the value of the pseudorandom function at location $i \cdot v$ (i concatenated with v). This prevents the adversary from "making up" values for memory locations, but it does not prevent the write-once (or replay) attack. That is, the adversary might continue to return old, obsolete value, tag pairs from a location. A similar problem was addressed in [7] and [15]. We address the problem in section 5.1.2.

3.3 Universal one-way hash functions

The advantage of This technique assumes only a reliable but not secret memory for the checker.

Let U be a family of functions where $\forall f \in E$, $f : D \mapsto R$. Following Naor and Yung [14] we say that U is a family of *universal one-way hash functions* (UOWHF) if $\forall x \in D$, for f chosen at random from U it is hard to find $y \neq x$ such that $f(x) = f(y)$ (see exact definition below). It is possible to construct a family of UOWHF given any one-way function ([14] shows this for any 1-1 one-way function and Rompel [17] shows this for any one-way function.)

Using UOWHF, there is a way to authenticate several memory cells with one memory cell, without assuming secrecy (but assuming that the contents of the authenticating cell were not altered). Let E be a family of UOWHF such that $\forall f \in U$ $f : D^2 \mapsto D$. Assume that a description of $f \in E$ is stored in the reliable (but not secret) memory of the checker. To authenticate values v_i and v_j, store in another cell (say l) $f(v_i, v_j)$. Assume that this l's contents have been verified somehow. Then, in order to verify the content of cell i or j: read the other cell; compute $f(v_i, v_j)$; and compare with the content of l.

We now define UOWHF precisely. Let $\{n_{1_i}\}$ and $\{n_{0_i}\}$ be two increasing sequences such that for all i $n_{0_i} \leq n_{1_i}$, but $\exists q$, a polynomial, such that $q(n_{0_i}) \geq n_{1_i}$ Let H_k be a collection of functions such that for all $h \in H_k$, $h :\ \{0,1\}^{n_{1_k}} \mapsto\ \{0,1\}^{n_{0_k}}$ and let $E = \bigcup_k H_k$. Let $\mathcal{A}$ be a probabilistic polynomial time algorithm ($\mathcal{A}$ is the collision adversary) that on input k outputs $x \in\ \{0,1\}^{n_{1_k}}$ which we call an *initial value*, then given a random $h \in H_k$ attempts to find $y \in\ \{0,1\}^{n_{1_k}}$ such that $h(x) = h(y)$ but $x \neq y$. In other words, upon receiving h, $\mathcal{A}$ tries to find a collision with the initial value.

Definition: Such a family U is called a *family of universal one-way hash functions* if for all polynomials p and for all polynomial time probabilistic algorithms $\mathcal{A}$ the following holds for sufficiently large k.

1. If $x \in\ \{0,1\}^{n_{1_k}}$ is $\mathcal{A}$'s initial value, then

$$\text{Prob}[\mathcal{A}(h, x) = y, h(x) = h(y), y \neq x] < \frac{1}{p(n_{1_k})}$$

   where the probability is taken over all $h \in H_k$ and the random choices of A.

2. $\forall h \in H_k$ there is a description of h of length polynomial in n_{1_k}, such that given h's description and x, $h(x)$ is computable in polynomial time.

3. H_k is accessible : there exists an algorithm G such that G on input k generates uniformly at random a description of $h \in H_k$.

For our purposes we will need a family with the following parameters: $n_{0_k} = k$ and $n_{1_k} = 2(k + \lceil \log |H_k| \rceil)$. That is $h \in H_k$ maps two strings of length k and two descriptions of elements of H_k into a string of length k. From [14] we know that a construction of such a family can be obtained, given any construction of a family that compresses one bit (which in turn can be based on any one-way function). The construction is such that $\lceil \log |H_k| \rceil$ is $O(k)$.

4 Off-line checkers

We adopt the same basic strategy in designing off-line checkers for RAMs, stacks, and queues. In its private memory the checker holds the following pieces of information:

- The description of a hash function h.
- The hashed value $h(W)$ of a string W that encodes the information in all the write instructions to the data structure.
- The hashed value $h(R)$ of a string R that encodes the information in all the read instructions to the data structure.

We will choose the encodings such that $W = R$ iff D functions correctly. The particular choice of encoding will depend on the data-structure being checked.

We have several constraints on the hash function h:

1. The description of h, $h(R)$, and $h(W)$ must all fit in the checker's memory.
2. We must be able to quickly update $h(W)$ and $h(R)$.
3. If $W \neq R$ then $h(W)$ must differ from $h(R)$ with high probability.

We now describe how we achieve these goals for the encoding and the hash function in each of the three data structures.

4.1 Checking RAMs

To check that a RAM operates correctly we must check that the value we obtain from reading an address is the last value previously written to that address. To perform this check we store in each memory address not only a value but also the time the value was written. Here 'time' is discrete and incremented whenever a write operation is performed on the data structure. The set of (value, address, time) triples which are written should equal the set of (value, address, time) triples which are read. The strings W and R are designed to represent these sets.

One possible encoding of the triples to W and R is as follows. Suppose (v, a, t) is one of the triples that is written. We encode this as a 1 at position $v + an + tn^2$ in W where n is the size of the RAM available to the user.

Here we assume v, a, and t are $\log n$-bit words. Thus the strings W and R have polynomial length. These strings are too large to store explicitly in the checker memory. Instead, we store the hash of W and R using ϵ-biased hash functions. In this case, the description of an ϵ-biased hash function requires $O(\log n + k)$ bits. $h(W)$ and $h(R)$ are each k bits long.

We now describe the functioning of the checker on write and read requests from the user:

Checker on user Write of value v to address a

- reads the value v' and time t' stored in address a.

- checks that t' is less than the current time.
- updates the hash $h(R)$ of string R with v', a, t'.
- writes the new value v and current time t to address a.
- updates the hash $h(W)$ of string W with v, a, t.

Checker on user Read of address a

- reads the value v' and time t' from address a.
- checks that t' is less than the current time t.
- updates the hash $h(R)$ of string R with v', a, t'.
- writes v' and t to address a.
- updates the hash $h(W)$ of string W with v', a, t.

Updating $h(W)$ on a write triple (v, a, t) (for an ϵ-biased hash function) involves complementing bit i of $h(W)$ if bit $v + an + tn^2$ of the i^{th} distinguisher is 1. Determining any one bit of a distinguisher string requires $O(1)$ operations on $\log n$-bit words. Thus updating $h(W)$ which consists of the inner product of W with k distinguisher strings requires $O(k)$ operations. The same holds for updates to $h(R)$.

To check the functioning of the RAM at the end of a sequence of operations, the checker reads all the memory cells and updates $h(R)$ accordingly. Assuming initially $W = R = 0$ and the RAM is empty, $h(W)$ should equal $h(R)$ if the memory functioned correctly, and should be different from $h(R)$ with high probability if the memory was faulty. We can ensure that t is less than n by checking the memory every n operations and resetting the time.

To show the checking scheme works correctly we must prove:

Lemma 1 *If the RAM malfunctions then $W \neq R$.*

Proof: A write operation performed by the checker and a read operation performed by the checker *correspond* if the read is the first read after the write involving the same address as the write. The fact that both a read and write operation are performed for each user operation insures that each read/write operation performed by the checker has a corresponding write/read operation. A malfunction occurs if the value and time the checker reads from an address are different from the value and time of the corresponding write. Let (v, a, t) be the value, address, and time of a write operation whose corresponding read returns v', t' with either $v' \neq v$ or $t' \neq t$. Choose such a triple, (v, a, t), such that t is maximized. In other words, we pick a write operation, with a corresponding errant read, such that the time of the *write* is maximized.

We want to show that no read operation at address a returns v, t. If we show this then the lemma follows since the triple (v, a, t) is stored in W but (v, a, t) is not stored in R. Consider the operations involving address a as occurring on a time line. No read operations after the chosen errant read can return (v, t) for this would be an errant read whose corresponding write has a larger time stamp.

Suppose (v, t) were returned before the write corresponding to the errant read. If this were the case then t would be greater than the current time at that read which would be detected by the checker. ∎

We summarize the results in the following theorem

Theorem 1 *For a RAM with* $2n$ *memory locations storing* $\log n$*-bit words there exists an off-line, invasive checker which uses* $O(\log n + k)$ *private memory and detects errors with probability* $\geq 1 - 1/2^k$.

4.2 Checking stacks

The same scheme used to check RAMs can be used to check stacks. The "address" of a stack operation is the level of the stack which is kept in checker memory. On a push operation we push the value and time onto the stack and update $h(W)$. Note that the level at which the item is pushed is empty before the operation. Thus we do not need to update $h(R)$. On a pop operation we pop both value and time. We check that the time is less that the current time and update $h(R)$. Again we do not update $h(W)$ since the level of the pop is emptied. With these modifications the above theorem for RAMs applies to stacks as well.

We can reduce the invasiveness of the checker by taking advantage of the restricted data access pattern of the stack. Rather than maintain the current time, the checker maintains the number of times the stack level achieves a local minimum. In other words, the checker counts the number of times the stack "turns around" after a sequence of pops and starts a sequence of pushes. The checker uses this count of local minima in place of the time stamp. The count, like the time, is strictly increasing for any particular level. Thus the proof that $W \neq R$ if an error occurs holds when time is replaced by count. The count remains the same during a sequence of pushes. We reduce the invasiveness by pushing the count only on the first push operation after a sequence of pops. The amount by which we reduce invasiveness is dependent on the input sequence, but this scheme is always less invasive than the scheme based directly on the RAM checker.

4.3 Checking queues

The RAM checker can also be used to check queues. In this case, the "address" of a write operation is the number of preceding write operations and the address of a read operation is the number of preceding read operations. An address in this sense is never reused thus the time stamp is redundant.

Let w be the number of values enqueued and r the number of values dequeued. The checker maintains w and r in private memory. On an enqueue of value v, the checker updates $h(W)$ with v and w, enqueues v, and increments w. Note that the checker is noninvasive. It writes only the input value v to the queue. On a dequeue, the checker dequeues some value v', updates $h(R)$ using v' and r, and increments r.

The queue malfunctions if and only if the i^{th} value dequeued does not match the i^{th} value enqueued for some i. Thus, as in the RAM case, after the queue is empty, $W \neq R$ if a malfunction occurs.

5 On-line checkers

The previous sections contain descriptions of checkers which check whether a *sequence* of operations are correct. In this section we describe checkers which check after *each* operation whether the data structure performed correctly.

5.1 Checking RAMs

It is possible to check a RAM using either pseudorandom functions or UOWHF. The overhead is $O(t \log n)$ where t is the time to evaluate the pseudorandom function or UOWHF and the memory size is $O(n)$.

Both solutions construct a complete binary tree on top of the memory. The leaves of the tree correspond to the n locations in the memory which are available to the user. The checker uses the remaining memory to store the internal nodes of the tree. Thus the checkers in both cases are invasive. To authenticate location i, all the nodes on the path from the root to location i and their children are accessed ($2 \log n$ altogether).

In the case of the pseudorandom function based checker, the checker's reliable memory must be secret, while in the case of the UOWHF based checker, the checker's reliable memory is known to the adversary. In the case of the UOWHF, if the word size of the RAM is $O(\log n)$ then since the adversary knows the checker's hash function, a polynomial time adversary could hash all $O(\log n)$ size words and find a value which collides with some value in the input sequence. In order to defeat such an adversary, the word size of the RAM must be polynomial. This is the primary difference between the two cryptographic based checkers.

5.1.1 Authentication using UOWHF

Let U be a family of UOHWF as defined in Section 3.3. For any $\epsilon > 0$ we can choose k, the parameter that determines the hashing domain, to be n^{ϵ}. In practice we should assume that k is sufficiently large so as to prohibit any feasible adversary $\mathcal{A}$ from breaking H_k.

We operate in words size k. The n locations which we wish to authenticate are assumes to be of size k. Each internal node v stores $k + \lceil \log |H_k| \rceil$ bits, $\lceil \log |H_k| \rceil$ bits that describe a function $h_v \in H_k$ and k bits that describe $x_v \in \{0,1\}^k$, the result of applying h_v to the contents of the children of v. The checker's memory is the root of the tree. It need not be kept secret.

In order to read the contents of location i, all the nodes on the path from the root to location i and their children ($2 \log n$ altogether) are accessed. For each internal node v on the path and its two children u and w verify that $x_v = h_v(x_u \circ h_u \circ x_w \circ h_w)$. $\circ$ denotes concatenation and if u and w are leaves, then h_u and h_w are simply the all zero string.

In order to write a new value to location i, all the nodes on the path from the root to location i and their children are accessed. For every node v along the path a new hash function $h'_v \in H_k$ is chosen to replace h'_v. The value $x'_v = h'_v(x_u \circ h_u \circ x_w \circ h_w)$ is computed and stored in v. u and w are v's children x_u, h_u, x_w and h_w denote the new contents of u and w.

Therefore in order to read or write $2 \log n$ cells are accessed, each cell is $O(k)$ bits. The amount of computation needed is $\log n$ applications of the a hash function. This scheme can be seen as a variant of Merkle's tree authentication scheme for digital signatures [12]. The signature scheme in [14] is based on it as well.

To define security in our context we must specify the power of an adversary that attempts to attack our scheme. We assume that the adversary $\mathcal{B}$ is a probabilistic polynomail time machine that controls both the input and the memory. $\mathcal{B}$ adaptively selects a sequence of read and writes instructions. The checker performs every read and write as defined above. Whenever the checker accesses a memory cell (except the root of the tree which is stored in the checker's own memory), $\mathcal{B}$ can choose any value to return, as explained in Section 2. The checker is supposed to announce "BUGGY" if the result of the read operation is not what it should be. We say that $\mathcal{B}$ is successful, if a faulty result from a read operation is returned without the checker eventually announcing "BUGGY". For any adversary $\mathcal{B}$ and memroy size n we can define its probability of success which is taken over $\mathcal{B}$'s internal coin flips. For the scheme to be secure means that for any adversary $\mathcal{B}$ and any polynomial p, for sufficiently large n the probability that $\mathcal{B}$ succeeds is smaller than $\frac{1}{p(n)}$.

Theorem 2 *The above scheme is secure.*

Proof: In order to prove that the scheme is secure, we must show that an adversary $\mathcal{B}$ which successfully avoids detection can be used to break the UOWHF in polynomial time, which contradicts our assumption of the existence of UOWHF. Suppose that there is a probabilistic polynomial time adversary $\mathcal{B}$ such that $\mathcal{B}$ has a non negligible probability of deceiving the scheme described above. Whenever $\mathcal{B}$ is successful, we know that at some earlier point $\mathcal{B}$ must have given for the first time false contents for some node u. Let v be u's parent and w be u's sibling. Since we assume that u was the first node whose contents were altered, the scheme did get the right value for h_v and x_v. Since $\mathcal{B}$ was not detected, it must have claimed the contents of u to be h'_u, x'_u and the contents of w to be h'_w, x'_w such that

$$x_v = h_v(x_u \circ h_u \circ x_w \circ h_w) = h_v(x'_u \circ h'_u \circ x'_w \circ h'_w)$$

We will now use $\mathcal{B}$ to construct an advesray $\mathcal{A}$ that attacks U, the presumed family of UOWHF. $\mathcal{A}$ guesses the first point in time and the first node u where $\mathcal{B}$ will lie (it has a non negligible chance of guessing correctly). $\mathcal{A}$ simulates $\mathcal{B}$ up to the point where $\mathcal{B}$ is suppose to lie. The real contents of u and its sibling w, x_u, h_u, x_w, h_w, are output by $\mathcal{A}$ as the x on which it would attempt to break the one-way hash function. Now $\mathcal{A}$ is given a random hash function $h \in H_k$ on which it should find a $y \neq x$ such that $h(x) = h(y)$. $\mathcal{A}$ writes h as h_v (From $\mathcal{B}$'s view this is similar to what the chexker does). If indeed $\mathcal{B}$ comes up with h'_u, x'_u, h'_w, x'_w such that $h_v(x_u \circ h_u \circ x_w \circ h_w) = h_v(x'_u \circ h'_u \circ x'_w \circ h'_w)$, then $\mathcal{A}$ outputs $y = x'_u \circ h'_u \circ x'_w \circ h'_w$. In this case $\mathcal{A}$ succeeds. Therefore, if $\mathcal{B}$ has a non negligible chance of deceving the checker, then $\mathcal{A}$ has a non negligible chance of breaking the UOWHF. ■

5.1.2 Authentication using pseudorandom functions

We describe how to use pseudorandom functions and time stamps to authenticate the memory.

Let T be a pessimistic estimate on the total number of operations in the system and let V be the set of values that can be written in a memory word. Let D, the domain of the pseudo-random function, be $V \times \{1..2n-1\} \times \{1..T\}$ and let $\ell(n)$ be such that $2^{-\ell(n)}$ is a probability of deception that is tolerable. In location i of the memory, three items are stored: v_i, the contents of location i; t_i, the last time it was written; and $f_S(v_i, t_i, i)$, an authentication tag which prevents the adversary from "making up" values for location i. The problem is to ensure that t_i is indeed the last time location i was written.

To solve this problem we construct a complete binary tree where the leaves correspond to the n memory locations. We associate a timestamp with every node of the tree. The timestamp associated with a leaf i is t_i. The timestamp t_v associated with an internal node v with children u and w is $t_u + t_w$. The values in the internal nodes are authenticated as above, by storing in node v the weight t_v and $f_S(0, t_v, v)$ (the nodes of the tree are mapped to $\{1..2n-1\}$). The root of the tree is kept in the checker's private memory.

In order to read the contents of location i, all the nodes on the path from the root to location i and their children ($2 \log n$ altogether) are accessed. For each internal node v on the path, verify that t_v is indeed the sum t_u and t_w, the timestamp at v's children. The tags f_S should be verified as well.

In order to write into location i, all the nodes on the path from the root to location i and their children are accessed and the timestamp t_i is verified as above. Write v_i' (the new value), $t_i + 1$ and $f_S(v_i', t_i + 1, i)$ in the leaf corresponding to i. For each internal node v along the path, increase t_v by 1 and write $f_S(0, t_v + 1, 1)$.

Why is this immune against replays (write once)? The replay attack can only decrease the times stored in the tree. Since the root contains the true value, there must be a first point along the path where false values are retrieved from one or both siblings. However, the sum of these false values cannot be equal to those retrieved from the parent, since the false values are only smaller than the true ones.

Thus in order to return an incorrect value and escape detection, an adversary must guess the value of the pseudorandom function at some point correctly. Suppose that the checker has access to a truly random function, rather than a pseudorandom function. In this case an adversary $\mathcal{A}$ can do no better than guessing the value of the function. If $\mathcal{A}$ can choose a polynomial length input sequence such that against a pseudorandom based checker it has a non negligible chance of avoiding detection, then the adversary forms the basis of a polynomial time statistical test. Given a function, the test simply simulates the checker using the function against the adversary on the input sequence. If the adversary succeeds in avoiding detection then the test declares the function pseudorandom.

The techniques described in this subsection carry over to stack and queue checking. However, for stacks and queues, we can design on-line checkers that do not use any cryptographic assumption. We describe this in the next subsection.

5.2 Checking stacks

The stack checker described in section 4 checks correctness when the stack empties. One way to check correctness after each operation is to empty the stack after each operation, storing the contents in an auxiliary stack. The checker then checks that $h(W) = h(R)$ and refills the main stack from the auxiliary stack. It checks the auxiliary stack with two auxiliary hashes. Unfortunately, this could require $\Omega(T)$ operations per pop and $\Omega(T^2)$ operations total, where T is the number of operations in the request sequence.

The checker described in this section follows the auxiliary stack method. That is, the checker has access to two stack data structures. Both stacks may be controlled by a common adversary. One of the stacks is an auxiliary stack and is used to hold the contents of the other stack which we partially empty periodically. We note that the user could use both stacks. That is both stacks may contain user data. In this case each stack would act as the auxiliary stack for the other.

In order to avoid squaring the number of operations in order to check the behavior of the stack, the checker keeps intermediate "markers" in its private memory. A marker at level l is the value that $h(W)$ and $h(R)$ attained when the stack reached level l. The hash values $h(W)$ and $h(R)$ are reset after the marker is placed so that the marker above it is $h(W)$ and $h(R)$ for values above level l. Thus, when the checker checks an operation, it need only empty the stack down to the position of a marker and check that $h(W) = h(R)$. If the stack drops below a marker, the checker resets $h(W)$ and $h(R)$ to the values stored by the marker.

We use $O(\log H)$ markers and perform $O(\log H)$ amortized additional operations per user operation for this checker where H is the maximum number of items in the stack. The trick is the placement of the markers. We use an idea from the simulation by oblivious Turing machines of Pippenger and Fischer [16]. To simplify the explanation, we assume that we have $h = \log H$ stacks $S_0, S_1, ..., S_{h-1}$. Each stack has its own $h(W)$ and $h(R)$. We will see how to combine these stacks into one stack later. The capacity of stack S_i is 2×2^i words. It is convenient to think of stack S_i as holding two blocks each of size 2^i. A block operation on stack S_i is a sequence of 2^i pushes or 2^i pops. We refer to block operations for convenience. Of course, the actual operations performed on the stack involve single words.

The stacks act as buffers. We service push/pop operations using stack S_0. If S_0 overflows (i.e., S_0 contains two items and receives a push operation), we remove the two data items in S_0 and push them (as one block) into stack S_1. Similarly, if on a pop operation S_0 is empty, we pop two data items (one block) from S_1 and push them into S_0. S_0 now contains two items and it can service the pop request.

The operation of stack S_i is identical to S_0 except that S_i uses a block of 2^i words as its data item. A simple inductive argument shows that following this strategy stack S_i receives a block push/pop operation at most every 2^i user operations. The time (number of single word operations) S_i requires to service a block push/pop operation is $O(2^i)$. This includes the time to empty S_i (checking that $h(W) = h(R)$ for S_i) and refill S_i from the auxiliary stack (checking the auxiliary stack). Thus, the time to service n user operations is $O(n \log H)$. Note that since H is the maximum height of the stack, S_{h-1} never overflows.

To turn the stacks $S_0, S_1, ..., S_{h-1}$ into a single stack, we stack the stacks on top of each other; the contents of S_0 above the contents of S_1, etc. Note that we still need a separate auxiliary stack to perform the partial emptying and refilling at each step. We keep a $O(h)$-bit vector in reliable memory which indicates the number of blocks in each of the h stacks. This vector determines the position of the markers. Each marker is a pair of $O(k)$-bit hash values. Thus the size of the checker's reliable memory is $O(k \log n)$.

For queues, a similar on-line checker can be implemented with $O(\log n)$ queues. However, at this point we do not know a simple way to combine these queues into a single queue.

6 Lower bounds

In this section, we describe two lower bounds on checker memory size. The first is a lower bound that holds essentially for all types of checkers considered in this paper. The second is a much stronger lower bound specifically for on-line, noninvasive checkers. We also show that these lower bounds are tight.

6.1 Lower bound for off-line checking

We show lower bounds on the amount of private memory a checker must use to correctly check sequences which store n bits of data. We prove three claims. The first applies to checkers which never call an honest implementation BUGGY. The second applies to checkers which may sometimes call an honest implementation BUGGY. The third claim extends the second claim by allowing the checker to use a (size $dn : 0 < d < 1$) public incorruptible tape. In each claim, we show that the checker needs close to $\log(n)$ bits in its private memory to work correctly.

Our lower bound is constructed from the special scenario in which the sequence of operations to be performed is a sequence of writes to distinct addresses followed by a sequence of reads from those addresses. This is a possible scenario for all the data structures we consider in this paper. For convenience, we will think of the sequence of writes as storing a long string which the checker must reconstruct at a later point after the sequence of reads.

We view the data structure as a large adversarial *main memory* which is accessible to the checker. We allow this adversary to be very powerful. We give the adversary the ability to place the main memory in any configuration following the sequence of write operations by the checker. The adversary's primary limitation is that it doesn't know the input string or the state of the checker's memory.

The checker's input is the n-bit string of data presented in the sequence of writes. The checker encodes the input as a checker memory state and a main memory state. Then after the sequence of reads the checker must reconstruct the input string using only the contents of its checker memory and the adversarial main memory.

We show that if the checker's memory is too small then there exists an input x such that the adversary can fool the checker when the checker tries to encode and decode x. By fooling the checker, we mean that the adversary changes the main memory contents such that the checker decodes the main memory and its checker memory as some y not equal to the input x. The adversary does not know what the checker's input is. However, the adversary only needs to be able to fool the checker (with high probability) on one input x in order to defeat the checker. Therefore, the adversary will always assume that the checker's input is x. If it is x, then the checker will be fooled. In the first claim, we also show that if the input is not x, the adversary might not fool the checker but it will escape detection – that is, even though the adversary alters main memory, the checker will not say BUGGY. It is easy to extend the two subsequent claims so that the adversary always escapes detection.

Claim 1 *A checker which correctly decodes the string it stored if the main memory is honest must have private memory of size* $m \geq \log(n) - 1$ *where* n *is the length of the string.*

Proof:

Assume $m < \log(n) - 1$ and fix the checker protocol.

We will show that there exists an input string x of length n such that whenever the checker encodes x as some main memory state and checker memory state and then tries to reconstruct the input, an adversary may always substitute a different main memory state and fool the checker into believing that the original input was something other than x.

The adversary considers how the checker encodes inputs and decodes combinations of checker memory and main memory states.

Definition: For each input x and each main memory state M, let $sphere(x, M)$ be a vector of length 2^m where the c^{th} component is set to A if the (M, c) pair is a possible encoding of x, $*$ if the pair is a possible encoding of some other input, and I if it is impossible for the checker to reach that pair from any input.

If the adversary does not alter the main memory (i.e. the main memory is honest) the checker must decode the main memory and checker memory as the input x with probability 1. Thus on input x, the checker memory must be in some configuration c and the main memory in some configuration M such that the c^{th} component of $sphere(x, M)$ is A.

Observation: If $\exists x$ such that $\forall M$, $\exists y \neq x$ and M' such that $sphere(x, M) = sphere(y, M')$ then the adversary can always fool the checker into believing it has stored $y \neq x$. Note that y may depend on M.

The adversary does this by always substituting M' for M (even though the adversary does not know what the checker's input actually was). If the input was x and the checker memory was c, then the c^{th} component of $sphere(x, M) = sphere(y, M')$ is an A and the checker will decode (M', c) as y.

Note also that if the input to the checker is not x and the main memory is M, then the checker memory c must correspond to a $*$ component of $sphere(x, M) = sphere(y, M')$ and the checker will still decode the memory pair (M', c) as some input and the adversary will go undetected.

In order to foil the adversary, an input must have a sphere (corresponding to that input and some main memory) which occurs for no other inputs. There are 2^n inputs of length n and there are only 3^{2^m} unique spheres. If $m < \log(n) - 1$ then $3^{2^m} < 2^n$. Hence there exists an input which satisfies the condition and which the checker cannot safely store. ■

Now assume that the checker functions correctly with probability p but with two-sided error. By this we mean that if the main memory is untouched by the adversary, the checker need only correctly decode the checker memory and main memory states with probability $\geq p$ and if the adversary alters the main memory, the checker will either detect the cheating or still correctly decode with probability $\geq p$.

Adding discretization to the techniques of the above proof, we show:

Claim 2 *A checker which functions correctly with probability $p \geq 1/2 + 1/2^{l+1}$ on input sequences storing n bits of data must have private memory of size $m \geq \log(n) - \log(l)$ where $l \in Z^+$.*

In other words, if the checker uses $\log(n)$ minus a few bits of checker memory, then there exists an input such that the probability that the checker can correctly decode that input is at most something close to $1/2$.

Proof:

Assume $m < \log(n) - \log(l)$. The adversary will look at spheres from the point of view of how the checker decodes the main memory, checker memory pair.

Redefine $sphere(x, M)$ to be a $l2^m$-bit vector where the c^{th} set of l bits is the closest binary approximation to Pr[checker decodes (c, M) as x].

Note that now there are at most $2^{l2^m} < 2^n$ unique spheres and $\exists x$ such that $\forall M, \exists y \neq x$ and M' such that $sphere(x, M) = sphere(y, M')$. Fix such an x.

Let

$$q = \sum_{M,c} (\Pr[\text{checker generates } M, c \text{ from } x] \times \Pr[\text{checker decodes } M, c \text{ as } x])$$

In other words, q is the probability the checker successfully encodes and decodes x when the adversary does not alter the main memory.

Let

$$q' = \sum_{M,c} (\Pr[\text{checker generates } M, c \text{ from } x] \times \Pr[\text{checker decodes } M', c \text{ as } y])$$

where $sphere(x, M) = sphere(y, M')$. In other words, q' is the probability that if the checker encodes x, and the adversary alters the main memory by finding a matching sphere then the adversary successfully fools the checker into decoding the checker memory and altered main memory as some other input.

Since $sphere(x, M) = sphere(y, M')$ and the approximation of each vector component uses l bits, Pr[checker decodes M, c as x] $<$ Pr[checker decodes M', c as y] $+ 1/2^l$.

Thus $q - 1/2^l < q'$.

By the definition of our checker $p < q$ and $q' < 1 - p$ which implies that $p < 1/2 + 1/2^{l+1}$.

■

Note that, if the checker uses too few bits in its private memory, then the probability an adversary can fool a checker is almost as high as the checker's probability of successfully decoding from unaltered memory ($q' > q - 1/2^l$). Thus we need only insist that the checker successfully decode unaltered memories with high probability in order to show that an adversary can fool the checker with high probability.

We now consider the case where the checker is allowed a certain amount of reliable memory which the adversary may also see. If the size of this public incorruptible memory were n then the checker could simply store the input in this memory. We consider the case where the size of the incorruptible memory is less than n by a constant fraction and show that the checker still needs $\Omega(\log(n))$ private memory.

Claim 3 *A checker which functions correctly with probability $p \geq 1/2 + 1/2^{l+1}$ on input sequences storing n bits of data using a public incorruptible tape of size dn ($d < 1$) must have private memory of size $m \geq \log(n) - \log(\frac{l}{1-d})$ where $l \in Z^+$.*

Proof:

Assume $m < \log(n) - \log(\frac{l}{1-d})$.

Define spheres as in the previous proof except now allow the spheres to be a function of the input x, the main memory M, and the incorruptible memory B. Using the same discretization as in the previous proof, we have that there are at most 2^{l2^m} unique spheres per incorruptible memory configuration. There are 2^{dn} possible incorruptible memory configurations and hence at most $2^{l2^m}2^{dn} < 2^n$ inputs have a unique sphere for any of the possible incorruptible memory configurations. Thus $\exists x$ and B such that $\forall M, \exists y \neq x$ and M' such that $sphere(y, M', B) = sphere(x, M, B)$.

Therefore, an adversary may always replace M with M' and the overall probability the checker will be fooled will be at least $p - 1/2^l$ if the checker's input were x. As in the previous proof this implies that $p < 1/2 + 1/2^{l+1}$. ■

6.2 Time-space tradeoffs for on-line noninvasive checking

We show a time-space tradeoff in the case of on-line noninvasive RAM checkers. Time in this context is t, the number of cells in the RAM examined by the checker when it checks the validity of an operation. Space is the size m of the checker's reliable memory. Let n be the size of the RAM. We show that $n \in O(mt)$. The proof is a generalization of Yao's tradeoffs for coherent functions in [18].

For the sake of simplicity we assume that each RAM cell holds just 1 bit. Once again we assume that the checker is correct with probability p whenever it certifies the contents of a memory location. Clearly, the interesting case is when $p > 1/2$.

Let M be the contents of the RAM and R an r-bit string which is treated as the source of randomness for the checker. Since the checker in noninvasive, all memory contents are possible and the number of pairs $\langle M, R\rangle$ is 2^{n+r}. The idea of the proof is to show that we can use an on-line noninvasive checker to encode the pair $\langle M, R\rangle$ as a smaller string from which the original pair can be reconstructed. If we can perform this encoding for too many pairs, we obtain a contradiction.

In particular, we show that at least a constant fraction γ of the pairs $\langle M, R\rangle$ can be uniquely specified by strings of length $m + (t+\beta)n/(t+1) + r$ with $\beta < 1$. It follows that,

$$\begin{aligned} n + r + \log\gamma &\leq m + (t+\beta)n/(t+1) + r \\ (1-\beta)n &\leq (m - \log\gamma)(t+1) \\ n &\in O(mt) \end{aligned}$$

The encoding of $\langle M, R\rangle$ is $\langle C, M', Z', R\rangle$ determined by the following steps:

1. Simulate the storage of M into the RAM using R as the source of randomness in order to obtain the checker memory C.

2. Let $J = \emptyset$, $j = 0$, $M' = \emptyset$

3. Repeat the following $n/(t+1)$ times

    Select the smallest positive integer $i \notin J$

    Assume $M_i = 0$ and check (reading t locations determined by the checker using C and R)

    If checker says BUGGY assume $M_i = 1$

    (Note: At this point we have assumed either $M_i = 0$ or $M_i = 1$.)

    If the assumption is correct then $Z_j = 1$ else $Z_j = 0$

    $j = j + 1$

    Concatenate the contents of the t locations read by the checker to M'

    Put i and the t locations read by the checker in J

4. If at least $\alpha n/(t+1)$ of the Z_j's are 1 then output $\langle C, M', Z', R\rangle$ where Z' and α are defined below.

The probability that the algorithm makes the correct assumption for the value of M_i is $\geq p$. The expected number of 1's in Z is then $\geq pn/(t+1)$. By Markov's inequality, at least $\gamma = (p-\alpha)/(1-\alpha)$ of the strings R cause Z to have $\geq \alpha n/(t+1)$ 1's. Since this holds for every memory contents, at least γ of the pairs $\langle M, R\rangle$ output an encoding $\langle C, M', Z', R\rangle$.

Given C, M', Z, and R we can clearly reconstruct M and R. We now show how to compress Z ($n/(t+1)$ bits) to Z' ($\beta n/(t+1)$ bits with $\beta < 1$) without losing any information. We know that Z contains $\geq \alpha n/(t+1)$ 1's. Using Chernoff bounds, the probability that a randomly chosen $n/(t+1)$-bit string has $\geq \alpha n/(t+1)$ 1's is $\leq e^{-(\alpha-1/2)^2 n/(2(t+1))}$. Thus the number of such strings is $\leq 2^{\beta n/(t+1)}$ where $\beta = 1 - \log(e)(\alpha - 1/2)^2/2$. We can encode these strings using $\beta n/(t+1)$ bits. Thus the length of $\langle C, M', Z', R\rangle$ is $m + (t+\beta)n/(t+1) + r$. The definition of the checker allows $p = 3/4$, $\alpha = 5/8$, $\gamma = 1/3$, and $\beta = 1 - \log(e)/128$. It follows that $n \in O(mt)$.

7 Conclusions and open problems

In this paper, we extend the idea of function checking to the realm of data structures and memory. We define this notion of memory checking and consider models in which the power of the checker is restricted in various ways (on-line/off-line and invasive/noninvasive). We present checkers for several data structures in these models. We show lower bounds on the amount of reliable memory a checker must use under very general assumptions. We also present a time-space tradeoff for a RAM checker under certain restrictions on its power (on-line, noninvasive). Here time measures the number of memory cells examined per operation and space measures the size of the checker's reliable memory. The checking model in this case closely resembles the setting for Boolean function coherence. Such a result indicates that data structure checking provides a framework for generalizing more traditional function checking models.

Another area in which data structure checking provides insight is in the context of interactive proofs. One application of our results is a direct method of simulating a polytime verifier by a logspace verifier. Condon [5], Dwork and Stockmeyer [6] and Lipton [10] derive this result to show that logspace verifiers can verify essentially the same proofs as polytime verifiers. Given a polytime verifier V (a Turing machine), we simulate V by a logspace verifier V'. V' maintains V's head position on its logspace worktape. It uses the prover P to hold the contents of V's tape. In this context, P plays the role of an unreliable memory accessed by tape position. Our results show that V' needs only logspace in order to check that P does not corrupt the tape. The computation time taken by V' is essentially that required by V. An interesting open problem is to discover further relations between memory checking and interactive proofs along with other areas of complexity theory.

An obvious open problem is to discover checkers for more complicated data structures. For example, checkers for heaps or binary search trees. These data structures have a more complicated structure than RAMs, stacks, or queues since their structure depends on the values of the data they contain. The structures studied in this paper all have the property that the value of the data did not play a part in determining how the data was stored. Either the order of operations or a specific address determined the data requested. In the case of heaps or binary search trees, the value of the data determines what value a read operation should return.

Another question is to study how invasiveness increases the power of data structure checkers. An interesting result would be to show invasiveness lower bounds for checking RAMs or stacks, or alternatively to show that invasiveness is not needed. A related problem is to reduce the overhead introduced by the checker per user operation.

References

[1] L. Adelman, M. Huang, and K. Kompella, Efficient checkers for number-theoretic computation. preprint.

[2] M. Blum, W. Evans, P. Gemmell, S. Kannan and M. Naor, Checking the correctness of memories, *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, pages 90–99, 1991.

[3] M. Blum and S. Kannan, Designing programs that check their work, *Proc. 21st ACM Symposium on Theory of Computing*, pages 86–97, 1989.

[4] M. Blum, M. Luby, and R. Rubinfeld, Self-testing and self-correcting programs with applications to numerical problems, *Proc. 22nd ACM Symposium on Theory of Computing*, pages 73–83, 1990.

[5] A. Condon, Space bounded probabilistic game automata, *Journal of the ACM*, 38(2), pages 472–494, April 1991

[6] C. Dwork and L. Stockmeyer, Finite state verifiers I: the power of interaction, *Journal of the ACM*, to appear.

[7] O. Goldreich, Towards a theory of software protection and simulation by oblivious rams, *Proc. 19th ACM Symposium on Theory of Computing*, pages 182–194, 1987.

[8] O. Goldreich, S. Goldwasser, and S. Micali, How to construct random functions, *Journal of the ACM*, 33(4):792–807, October 1986.

[H] J. Hastad, *Pseudo-random generators under uniform assumpsions*, Proc. 22nd Symposium on Theory of Computing, 1990.

[9] R. Impagliazzo, L. A. Levin and M. Luby Pseudo-random generators form one-way functions, *Proc. 21st ACM Symposium on Theory of Computing*, pages 12–24, 1989.

[10] R. Lipton, Efficient checking of computations, *Proc. 7th Annual Symposium on Theoretical Aspects of Computer Science*, pages 207–215, 1990.

[11] R. Lipton, New directions in testing, preprint.

[12] R. Merkle. A certified digital signature, manuscript, 1979, see also Advances in Cryptology, Crypto '89 Proceedings, Lecture Notes in Computer Science 435, pp. 218–238, 1990.

[13] J. Naor and M. Naor, Small-bias probability spaces: efficient constructions and applications, *Proc. 22nd ACM Symposium on Theory of Computing*, pages 213–223, 1990.

[14] M. Naor and M. Yung, Universal one-way hash functions and their cryptographic applications, *Proc. 21st ACM Symposium on Theory of Computing*, pages 33–43, 1989.

[15] R. Ostrovsky, Efficient computation on oblivious rams, *Proc. 22nd ACM Symposium on Theory of Computing*, pages 514–523, 1990.

[16] N. Pippinger and M.J. Fischer, Relations among complexity measures, *Journal of the ACM*, 26(2):361–381, 1979.

[17] J. Rompel, One way functions are necessary and sufficient for secure signatures, *Proc. 22nd ACM Symposium on Theory of Computing*, pages 387–394 1990.

[18] A. Yao, Coherent functions and program checkers, *Proc. 22nd ACM Symposium on Theory of Computing*, pages 84–94, 1990.