Checking the Correctness of Memories*

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

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 (online) 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 [2] and several subsequent papers [3, 1, 7] have provided checkers for classical computational problems. In the context of program checking, programs for these problems have been thought of merely as computing a function and not as having any side effects.

An important problem is extending the concept 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.

The question of checking a sequence of stores and retrieves from a random access memory has been addressed by the papers of Goldreich[4] and Ostrovsky[11]. 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.

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.

*Supported in part by NSF grant CCR 88-13632

†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

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* 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 $> 3/4$.

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 it's 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 $O(\log n)$. 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 do not introduce operations to the data structure in addition to the user operations are called *noninvasive*, while checkers which do introduce operations in order to store checking information are called *invasive*.

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 [9]. 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 [9] 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 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

Assume that the reliable (and secret) memory of the checker can store the seed S of pseudorandom function f (*a la* Goldreich, Goldwasser and Micali[5]). Equivalently, assume that the checker is equipped with a private random oracle. 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, i.e., once a location has been authenticated, the adversary might stop any changes and this would go undetected without some other mechanism. A similar problem was addressed in [4] and [11]. We address the problem in section 5.1.2.

Since the construction of pseudorandom functions of [5] (the only one known) implies that the time to evaluate f is proportional to the (length of the argument) * (the time to pseudo generate a sequence twice as long as the seed), it makes sense to shorten the argument as much as possible. This can be done: Let T be a (pessimistic) estimator on the number of calls to f and let k be the checker's security parameter. Store in the checker's private memory a description of h, a random universal$_2$ hash function, where $h : D \mapsto \{1...T2^k\}$ and D denotes the domain of (address, value) pairs. The tag is now $f_S(h(i, v))$.

3.3 Universal one-way hash functions

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

Let E be a family of functions where $\forall f \in E$, $f : D \mapsto R$. E is a family of universal one-way hash functions (UOWHF) if $\forall x \in D$, for f chosen at random from E it is hard to find $y \neq x$ such that $f(x) = f(y)$ (see exact definition in [10].) It is possible to construct UOWHF given any one-way function ([10] shows this for any 1-1 one-way function and [13] shows this for any one-way function.)

Using UOWHF, it is possible to authenticate several memory cells with one, without secrecy. Let E be a family of UOWHF such that $\forall f \in E\ f : D^2 \mapsto D$. 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.

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 on 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.

Here we assume v, a, and t are $\log n$-bit words. Thus the strings W and R have polynomial length.

We use 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 of string R ($h(R)$) with v', a, t'.
- writes the new value v and current time t to address a.
- updates $h(W)$ 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 $h(R)$ with v', a, t'.
- writes v' and t to address a.
- updates $h(W)$ 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. This requires $O(1)$ operations on $\log n$-bit words. 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 functions correctly and 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 malfunction occurs if the value and time the checker reads from an address are different from the value and time which the checker previously wrote. Let (v, a, t) be the value, address, and time of a write operation whose corresponding (later) 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. 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 with a larger time stamp for the corresponding write.

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.

4.3 Checking queues

The RAM checker can also be used to check queues. In this case, the "address" of an operation is the number of values previously enqueued. 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 uses v and w to update $h(W)$ and enqueues v. On a dequeue, the checker dequeues some value v' and updates $h(R)$ using v' and r. The checker is noninvasive since it writes only the input values to the queue.

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 n is the memory size.

Both solutions construct a complete binary tree on top of the memory. The leaves of the tree correspond to locations in the memory. 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).

5.1.1 Authentication using UOWHF

We store in the checker's memory a description of h, the UOWHF. In each internal node, we store the value of h applied to both its children. We also store the value of the root in the checker's memory.

When storing a new value, we update the values stored in the internal nodes along the path from the root to leaf i accordingly.

When accessing location i, for each node along the path from the root to the leaf representing i, we verify that the value stored is h applied on the values of its two children.

This scheme can be seen as a variant of Merkle's tree authentication scheme for digital signatures[8]. The signature scheme in [10] is based on it as well. The proof that this scheme is secure against a resource bounded adversary follows from the definition of universal one way hash functions.

5.1.2 Authentication using pseudorandom functions

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

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. The value we associate with a leaf is t_i. Each internal node contains the sum of the values of its two children. The values in the internal nodes are authenticated as above (by f_S) but without the time they were written. The value at the root is kept in the checker's private memory.

To access location i, we access all the nodes on the path from the root to location i and their children ($2 \log n$ altogether). For each internal node, we verify that the value stored is indeed the sum of the values of its two children.

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.

Again with a suitable resource bound on the adversary we can prove that this scheme is immune against tampering.

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 this method except that it 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[12]. 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. 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. 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 total space used by the checker 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

Here, we consider some 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 a storing a long string which the checker must reconstruct at a later point after the sequence of reads.

For the sake of the lower bound we think of 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 it wants 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 its input as a checker memory state and a main memory state. Then on the sequence of reads the checker must reconstruct the original 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 tricks the checker into believing it stored an input different from the one it actually did. 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, 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 retrieve the input, an adversary may always substitute a different main memory state and deceive 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.

For each main memory state M, we define **ball**(M) to be a vector of length 2^m where the c^{th} component is what input value, if any, the checker could encode to reach the main memory, checker memory pair (M, c). By our assumption that for every input the checker encodes the input and decodes checker memories and honest main memories as that input with probability 1, we know that for all pairs M, c which are reachable by the checker on some input x the checker must decode that memory pair as x with 100% certainty. Therefore the c^{th} component of $ball(M)$ has at most one input value associated with it. For pairs of main memory and checker memory states which cannot be reached from any input, the adversary labels these vector components *impossible*.

We need the following definition:

Definition: For each input, x and each main memory state M, the **sphere**(x, M) is a vector 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.

Note that if the input were x and the main memory were M, the assumption that the checker works with probability 1 ensures that the checker memory must be in one of the configurations which correspond to an A component of $sphere(x, M)$.

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 was not x, 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 tricks 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 the tradeoffs for coherent functions in [14].

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}. 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 $i \notin J$

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

   If checker says BUGGY assume $M_i = 1$

   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. Lipton[6] derives 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. An interesting open problem is to discover further relations between memory checking and interactive proofs along with other areas of complexity theory.

References

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

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

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

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

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

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

[7] R. Lipton. New directions in testing. preprint.

[8] R. Merkle. A certified digital signature. manuscript, 1979.

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

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

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

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

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

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