Runtime Verification of Remotely Executed Code using Probabilistically Checkable Proof Systems *

Tuğkan Batu† Ronitt Rubinfeld‡ Patrick White§

Abstract

In this paper we consider the verification and certification of computations that are done remotely. We investigate the use of probabilistically checkable proof (PCP) systems for efficiently certifying such computations. This model can also be applied to verifying security proofs of software downloads. To make the use of PCPs more practical, a new version of Cook's Theorem is given for the RAM model: that is, we show that a correct computation of a RAM can be encoded as a satisfiable boolean formula. We use this result to show that the implementations of PCPs no longer need to be based on a description of the desired computation in terms of a Turing machine program.

1 Introduction

Remote execution of code is an attractive alternative to local execution, when powerful computing resources are available on a network such as the Internet. Yet the recipient of the result of a remote calculation needs to be able to trust that the answer computed is in fact correct. For example, one may be looking for the member in a combinatorial structure which is minimal under some function, such as the optimal tour of a graph, or might like to know that a certain boolean formula is unsatisfiable. Proving the correctness of a proposed answer to each can be performed by a simple exhaustive search, of all feasible tours in the former case, and all valid truth assignments in the second. Both lists are exponentially longer than the input to the problem, and hence than the length of the answer which is being transferred from the powerful computer. Certainly this type of verification is much too costly to be feasible. We consider the approach of applying probabilistically checkable proof systems (PCPs), which allow interactions between a simple and efficient verifying machine and a computationally powerful prover. PCPs provide a general mechanism by which verifier can, in approximately $\log T$ steps, trust the result of a computation requiring T time steps. This takes care of the exponential blow up in our above examples, reducing the time complexity to essentially that of reading the input. In fact, this model is advantageous in any situation in which the computation to be performed takes longer than reading the input. Thus it is even beneficial to use networked computing resources for solving quadratic time problems. Moreover, we can use this model to check the validity of a result without caring about how it is obtained, for example, verifying a satisfying assignment regardless of the process which produced it.

In this paper we consider a model in which a powerful untrusted machine can convince the user that the result of a computation is correct by applying PCP technology. Note that although conceptually the PCP protocols are quite intricate and their correctness is difficult to prove, the actual algorithms employed are not terribly complex. Nonetheless, one unpleasant complication is that all known constructions of PCPs require a description of the computation

*This work was partially supported by ONR N00014-97-1-0505, MURI, NSF Career grant CCR-9624552, and an Alfred P. Sloan Research Award.

†Department of Computer Science, Cornell University, Ithaca, NY 14850. email: batu@cs.cornell.edu.

‡Department of Computer Science, Cornell University, Ithaca, NY 14850. email: ronitt@cs.cornell.edu. Part of this work was done while on sabbatical at IBM Almaden Research Center.

§Department of Computer Science, Cornell University, Ithaca, NY 14850. email: white@cs.cornell.edu.

to be performed in the form of a Turing machine computation. Cook's Theorem is then applied to transform the computation into a satisfiable CNF formula. We instead show how to generate a CNF formula directly from a RAM (random access machine) description, thereby bypassing the need for a Turing Machine description. Even though it is a well-known fact that the RAM and Turing Machine models are equivalent, using the RAM model in this setting proves to be more practical because of the lack of a compiler from any random access model to Turing Machine model. From this point, the PCP protocol can continue unaltered. As a result we redescribe the PCP theorems in terms of a RAM model, which can quite easily be generated by a compiler from a modern high level language, such as C or ML.

1.1 The Model

We assume a computationally bounded user ("Verifier" or "V") is interacting with a very powerful, untrusted computer ("Prover" or "P") over a remote network. There are two specific models we consider. In the first, the Prover is to execute some program M on input x such that both V and P have access to $< M, x >$. Assume that M's computation time is T and that T is prohibitively large for the verifier. The prover generates a PCP proof to convince the verifier that M was executed correctly. It is the surprising result of [ALM+98] based on a long series of papers [LFKN90, BFL91, BFLS90, FGL+96, AS98] that the proof can be written in such a way that V can trust its correctness after inspecting only a constant number of locations in this proof. We use this machinery directly in our first model, which is depicted in Figure 1, in which the prover computes and writes down the proof. The verifier then determines which bits of the proof it wants to see, and the prover transmits these bits to V over the network. In previous PCP results it is assumed that M is given as a Turing machine. In this paper we show that M may be directly encoded as a RAM.

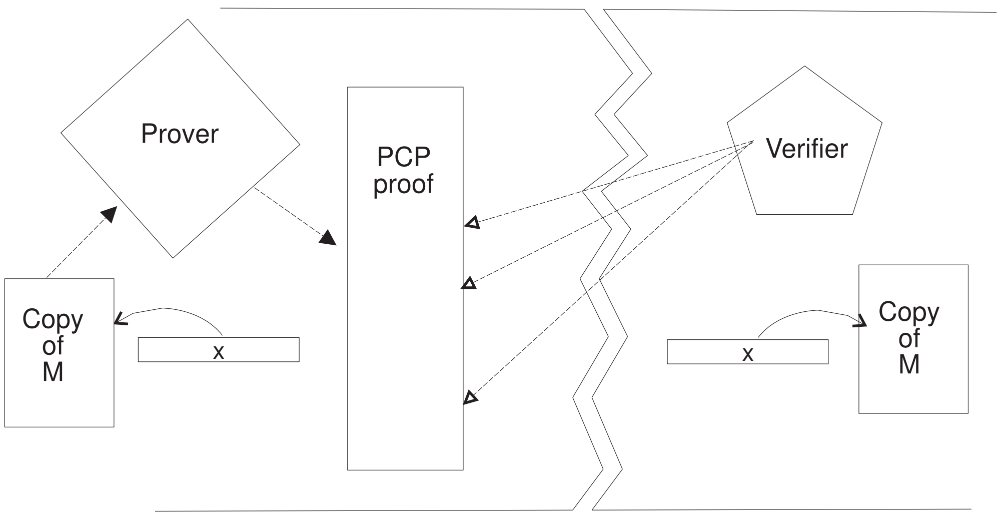

Figure 1: The Basic Model

The second model is an application to secure software download, which has been suggested previously by [DKL+]. V would like to download code (e.g. Netscape) and be convinced that the program has a certain property (e.g. this code will not crash). The first model can be specialized and extended to describe this type of interaction. The Prover provides a **Certificate** of a desired property along with a program. We do not care how the certificate is computed, but we expect that the certificate is so much larger than the code that downloading it is undesirable. The certificate may be very difficult to generate, yet it should be reasonably easy to deterministically certify its correctness. We now assume that both the Prover and Verifier have access to an agreed upon and relatively simple **Certifier**, which takes the program code and the certificate as input and accepts or rejects the certificate in time T.

(The Certifier is here playing the role of M in the first model. The copy of the program is playing the role of x). Without using PCPs, V would have to download the certificate and run the certifier on the certificate. Since the certificate can be large V can instead apply PCP techniques. The verifier spends $O(|x| + \log T)$ steps, where T is the number of steps required to run the certifier. To facilitate this, we again assume the certifier is compiled to RAM code. Using the same techniques as above, P runs the certifier M to check the certificate, and then encodes the run of M as a PCP proof. Now the Verifier acts exactly as in the first model, querying the proof in just a few locations to determine that the certificate is valid. This model is depicted in Figure 2.

As an example of how this model could be used, we will integrate the Proof-Carrying Code (PCC) technique ([NL96]) into our framework. In a PCC system, the code producer, which would be the prover in our model, provides a formal safety proof for a predefined safety policy. This safety proof will act as the certificate in our model. The certifier from our model will be replaced by the proof validator of the PCC system. This proof validator is a reasonably simple program which could be trusted by the code consumer (verifier). At this point, instead of uploading the whole certificate (safety proof) as it would in the PCC system, the code producer will produce and commit to the proof of the certificate being accepted by the certifier as described above. The proof validator is the part of the system that will be encoded as a RAM. The bandwidth of the communication will be reduced drastically as a result.

Proof Commitment One requirement of the PCP protocols is that the proof be written down by the prover and remain unchanged throughout the interaction of the prover and verifier. There are many ways in which the verifier can trust the proof remains unchanged (without downloading the proof). One possibility is to use a trusted third party: P transmits the proof to this third party, and the verifier interacts with the third party assuming that it has no reason to change the proof. Alternatively, one can force P to commit to the proof by using cryptographic techniques of [Mer90], as suggested by [Kil92] (also employed in CS proofs [Mic94] and the work of [DKL+]). This latter scheme introduces only a logarithmic overhead to the running time of the verifier. We assume one of these schemes is employed in what follows.

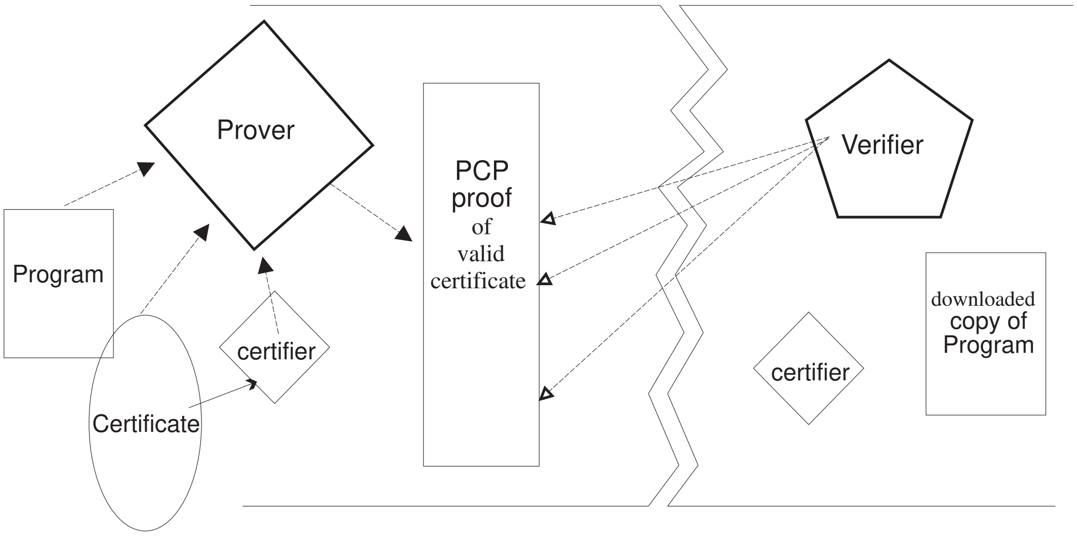

Figure 2: The Verification Model

High level overview of the PCP protocols The various PCP protocols described in the literature share a similar high level outline. To begin, a nondeterministic Turing Machine M is given, along with an input x of length n. The Turing Machine decides x, running in $T(n)$ steps. A prover turns the computation history of M into a 3CNF formula which is satisfiable if and only if M really accepts x. This formula is then transformed into a multivariate polynomial F whose zeros correspond in a one-one fashion to satisfying assignments of the 3CNF formula. The prover finds a value $\vec{x}$ such that $F(\vec{x}) = 0$, and then encodes this value in a special form. The vector $\vec{x}$ is so long that the verifier does not have time to even look at all of it. However, the encoded version of the proof provided by the prover enables the verifier to look at a small number of locations, verify some consistency properties, and if the verifications pass trust that $F(\vec{x}) = 0$ in $O(n + \log(T))$ time. From the existence of a zero, the verifier concludes that the 3CNF is satisfiable, and thus that M accepted x.

For our purposes, the version of PCP which is most applicable is a result of [BFLS90] which states the verification process in a theorem-proof model. For us the theorem is "This program has property Q," or "This program returns result x", and the proof is the certificate.

Theorem 1 *(***BFLS**[1]*) A theorem-proof pair $< T, P >$ can be probabilistically verified in time $O(|T| + \log |P|)$.*

Note that the number of locations that the verifier looks at the in proof can be made constant ([PS]) but the total communication overhead is still logarithmic, since the verifier needs to specify the addresses of the locations.

1.2 Our Results

The primary contribution of this paper is a direct reformulation of Cook's Theorem and hence the PCP characterizations of NEXPtime and NP in terms of RAMs instead of Turing Machines. Although it is known that RAMs and Turing machines are equivalent in power (cf. [Papa]), we show that a direct application of Cook's Theorem for RAMs is cleanly expressible and easily implemented. This makes it plausible that PCP technology can be employed in an algorithmic setting for realizing computational speed-ups, since it is no longer required that a Turing Machine be constructed to execute the program in question. The work of [BFLS90] shows that the machine M can be characterized in terms of Kolmogorov-Uspenskii machines. But again this would require that the prover implement M in terms of a Kolmogorov-Uspenskii machine, or have a compiler from a random access model to the Kolmogorov-Uspenskii model.

Next, we show how this technology can be applied in the setting of runtime result verification. The model we give is quite general and can be applied to the verification of any property which can be computed in nondeterministic exponential time. This question has also been considered in [Mic94, DKL+, FN].

2 Encoding a RAM by a Boolean Formula

2.1 The RAM model

A RAM as described in Papadimitriou [Papa] is a computing device which has direct (i.e. one-step) access to an unlimited number of registers $\{r_1, \ldots r_i, \ldots\}$, each of which can contain an arbitrarily long positive or negative integer. For our purposes, we will assume instead that these registers are space bounded and the largest location used is $S(n)$ which is a parameter of the machine. The input to a RAM is a tape containing a list $\{x_1, x_2, ..., x_n\}$ of integers, also with random one-step access. The result of a RAM computation is the contents of register 0 after the computation has completed. The RAM program itself is a sequence $\Pi = (\pi_1, \pi_2, ..., \pi_m)$ of any of the instructions given in Figure 3. Note that other desirable primitive operations, such as **mult**, **div**, **push**, **pop**, *etc.* can be easily simulated by the given operations, or added directly to the language in a way which will be clear from the exposition.

2.2 The Encoding

A correct computation of a RAM will be encoded as a satisfiable boolean formula. Each instruction of the RAM program is translated into a boolean sentence which will be quantified over all time steps of the execution of the

[1]This version depends on a linear time encoding of the theorem, due to [Spi96].

Instruction	Operand			Semantics		
read	j			$r_0 \leftarrow x_j$		
read	$\uparrow j$			$r_0 \leftarrow x_{r_j}$		
store	j			$r_j \leftarrow r_0$		
store	$\uparrow j$			$r_{r_j} \leftarrow r_0$		
load	j	$\uparrow j$	$= j$	$r_0 \leftarrow r_j$	$r_0 \leftarrow r_{r_j}$	$r_0 \leftarrow j$
add	j	$\uparrow j$	$= j$	$r_0 \leftarrow r_0 + r_j$	$r_0 \leftarrow r_0 + r_{r_j}$	$r_0 \leftarrow r_0 + j$
sub	j	$\uparrow j$	$= j$	$r_0 \leftarrow r_0 - r_j$	$r_0 \leftarrow r_0 - r_{r_j}$	$r_0 \leftarrow r_0 - j$
half				$r_0 \leftarrow r_0/2$		
jump	j			$\kappa \leftarrow j$		
jpos	j			**if** $r_0 > 0$ **then** $\kappa \leftarrow j$		
jzero	j			**if** $r_0 == 0$ **then** $\kappa \leftarrow j$		
jneg	j			**if** $r_0 < 0$ **then** $\kappa \leftarrow j$		
halt				$\kappa \leftarrow 0$		

j, an integer	r_j, contents of register j
r_{r_j}, contents of register r_j	κ, program counter

Figure 3: Instruction set of a RAM

RAM. The resulting formula will describe consistency conditions which must be true every time this instruction is executed. For example, for the `half` instruction we will ensure that each time it is executed, the contents of the accumulator do become one half of what they were before the instruction was initiated. In addition to boolean formulas that correspond to the individual instructions, we will also construct formulas that encode the consistency of the machine state throughout the computation. For example we ensure that at any time, each register has exactly one value stored in it. Lastly, the initial and final states of the machine will be encoded by another boolean formula.

Before giving the formulation, we would like to stress the fact that the resulting formula is big ($O(T^3)$). Yet the verifier never needs to write it down during the PCP verification process. It only has to be able to generate very small pieces of the formula. These pieces, we will see, can be easily computed given the highly structured way in which we translate a RAM program into a boolean formula.

Figure 2.2 lists the boolean variables used in our formulation. Our formulation is based on a proof of Cook's Theorem found in [Kozen]. Our local consistency conditions are essentially from his exposition. Our control flow and arithmetic operation conditions are necessarily more complicated because of the differences in the Turing machine and RAM models. The underlying motivation is to consider *configurations* of the RAM, where each configuration lists the entire state of the machine. We define decision variables which describe whether certain relations hold in these configurations. For example the indexed variable Q_t^i is true if and only if the machine is executing line number i at time step t.

Q_t^i	At time t, $\kappa = i$ (program counter points to the i^{th} instruction).
$R_t^{i,x}$	At time t, $r_i = x$ (register i contains the value x).
$I^{i,x}$	$x_i = x,$ (the value of i^{th} input is x).
P_t	At time $t, r_0 > 0$.
N_t	At time $t, r_0 < 0$.
C_t^x	At time $t,$ carry register has the value x stored in it.

Figure 4: The Boolean Variables

Now we give the formulas that capture the initial state and consistency conditions of the computation. In the formulas, the variable t refers to time steps, the variables i, j are used for register and input indices, and the variables

x, y, z are used for the register and input values. Whenever variable t is quantified, it is quantified over the set $\{1, \ldots, T\}$ where T is an upper bound on the running time of the RAM on that input. We also assume that the registers of our machine are bounded in space by some value $\log(S)$. The variables x, y, z will be quantified over binary strings of length $\log(S)$.

Local Consistency Conditions

- The initial state of the RAM is captured by the following formula (I, a sequence of integers $\langle x_1, \ldots, x_n \rangle$, is the input to the RAM):

$$Q_0^1 \wedge \left(\bigwedge_i R_0^{i,0} \right) \wedge \left(\bigwedge_{1 \le i \le |I|} I^{i,x_i} \right) \wedge \left(\bigwedge_{1 \le i \le |I|} \bigwedge_{x,y,x \ne y} \neg I^{i,x} \vee \neg I^{i,y} \right)$$

  The four subformulas of the formula encode the facts that the program counter is initialized to the first instruction, all registers initially have value 0 stored in them, the input values are written in the input cells, and each input cell has exactly one value stored in it, respectively.

- At any time, the machine is executing exactly one instruction.

$$\bigwedge_t \left(\bigvee_i Q_t^i \right) \wedge \bigwedge_t \bigwedge_{i,j,i \ne j} \left(\neg Q_t^i \vee \neg Q_t^j \right)$$

  The left conjunction ensures that there is at least one instruction executed at any time, the right conjunction ensures that at most one instruction is executed.

- At any time, each register contains exactly one integer.

$$\bigwedge_t \bigwedge_i \left(\bigvee_x R_t^{i,x} \right) \wedge \bigwedge_t \bigwedge_i \bigwedge_{x,y,x \ne y} \left(\neg R_t^{i,x} \vee \neg R_t^{i,y} \right)$$

  As above, two subformulas encode the facts that each register contains at least and at most one value at any time, respectively.

- At time t, carry register contains exactly one integer and carry for the first bit is always zero.

$$\bigwedge_t \left(\bigvee_x C_t^x \right) \wedge \bigwedge_t \bigwedge_{x,y,x \ne y} (\neg C_t^x \vee \neg C_t^y) \wedge \bigwedge_t C_t^0[1]$$

  This one is similar to the register consistency above. The rightmost conjunction (over time) ensures that carry bit into the least significant digit of any arithmetical computation is 0. The bracket notation [1] refers to the first bit of this register. These bits are necessary for encoding arithmetic operands, and in our arithmetization of the SAT clause. More detail is given below, but for now we just gloss over this detail.

- Finally we define an acceptance condition, in keeping with the Turing machine model. A RAM will be defined to accept if and only if it halts and sets the program counter to zero and stores a zero value in its accumulator. Hence the corresponding formula is

$$\bigwedge_t \left(Q_t^0 \longrightarrow Q_{t+1}^0 \wedge R_t^{0,0} \right)$$

Note that we can generalize the notion of acceptance by defining the value returned by a RAM computation. This is defined as the value left in the accumulator after the `halt` instruction is executed. By modifying the above boolean formula, we can easily extend the technique given here to arithmetize the fact that machine M on input x returns value y.

Control Flow Instructions In the next table, we give the translations of the move and control flow instructions (all instructions but the arithmetic ones) into boolean formulas. The equality sign $A = B$, is the shortcut for the formula $((A \wedge B) \vee (\neg A \wedge \neg B))$. The translations in the table assumes that the instruction is the n^{th} instruction of a RAM program.

Instruction π_n	Boolean Formula
read j	$\bigwedge_t \bigwedge_x \left[Q_t^n \wedge I^{j,x} \rightarrow Q_{t+1}^{n+1} \wedge R_{t+1}^{0,x} \wedge \left(\bigwedge_{i\neq 0} \bigwedge_y R_t^{i,y} = R_{t+1}^{i,y}\right)\right]$
store j	$\bigwedge_t \bigwedge_x \left[Q_t^n \wedge R_t^{0,x} \rightarrow Q_{t+1}^{n+1} \wedge R_{t+1}^{j,x} \wedge \left(\bigwedge_{i\neq j} \bigwedge_y R_t^{i,y} = R_{t+1}^{i,y}\right)\right]$
read $\uparrow j$	$\bigwedge_t \bigwedge_{x,y} \left[Q_t^n \wedge R_t^{j,x} \wedge I^{x,y} \rightarrow Q_{t+1}^{n+1} \wedge R_{t+1}^{0,y} \wedge \left(\bigwedge_{i\neq 0} \bigwedge_z R_t^{i,z} = R_{t+1}^{i,z}\right)\right]$
store $\uparrow j$	$\bigwedge_t \bigwedge_{x,y} \left[Q_t^n \wedge R_t^{j,x} \wedge R_t^{0,y} \rightarrow Q_{t+1}^{n+1} \wedge R_{t+1}^{x,y} \wedge \left(\bigwedge_{i\neq x} \bigwedge_z R_t^{i,z} = R_{t+1}^{i,z}\right)\right]$
load j	$\bigwedge_t \bigwedge_x \left[Q_t^n \wedge R_t^{j,x} \rightarrow Q_{t+1}^{n+1} \wedge R_{t+1}^{0,x} \wedge (\bigwedge_{i\neq 0} \bigwedge_y R_t^{i,y} = R_{t+1}^{i,y})\right]$
load $\uparrow j$	$\bigwedge_t \bigwedge_{x,y} \left[Q_t^n \wedge R_t^{j,x} \wedge R_t^{x,y} \rightarrow Q_{t+1}^{n+1} \wedge R_{t+1}^{0,y} \wedge (\bigwedge_{i\neq 0} \bigwedge_y R_t^{i,y} = R_{t+1}^{i,y})\right]$
load $= j$	$\bigwedge_t \left[Q_t^n \rightarrow Q_{t+1}^{n+1} \wedge R_{t+1}^{0,j} \wedge (\bigwedge_{i\neq 0} \bigwedge_y R_t^{i,y} = R_{t+1}^{i,y})\right]$
jump j	$\bigwedge_t \left[Q_t^n \rightarrow Q_{t+1}^{j} \wedge (\bigwedge_i \bigwedge_y R_t^{i,y} = R_{t+1}^{i,y})\right]$
jpos j	$\bigwedge_t \left[Q_t^n \rightarrow \left((P_t \wedge Q_{t+1}^{j}) \vee (\neg P_t \wedge Q_{t+1}^{n+1}) \right) \wedge (\bigwedge_i \bigwedge_y R_t^{i,y} = R_{t+1}^{i,y})\right]$
jzero j	$\bigwedge_t \left[Q_t^n \rightarrow \left((Z_t \wedge Q_{t+1}^{j}) \vee (\neg Z_t \wedge Q_{t+1}^{n+1})\right) \wedge (\bigwedge_i \bigwedge_y R_t^{i,y} = R_{t+1}^{i,y})\right]$
jneg j	$\bigwedge_t \left[Q_t^n \rightarrow \left((N_t \wedge Q_{t+1}^{j}) \vee (\neg N_t \wedge Q_{t+1}^{n+1})\right) \wedge (\bigwedge_i \bigwedge_y R_t^{i,y} = R_{t+1}^{i,y})\right]$
halt	$\bigwedge_t \left[Q_t^n \rightarrow Q_{t+1}^{0} \wedge \bigwedge_{t'>t} \left(Q_{t'}^{0} \wedge \left(\bigwedge_x R_t^{0,x} = R_{t'}^{0,x}\right)\right)\right]$

Figure 5: Translations for move and control flow instructions

As an example, a **jump 13** instruction on the 7^{th} line of a RAM program is translated into

$$\bigwedge_t \left[Q_t^7 \rightarrow Q_{t+1}^{13} \wedge (\bigwedge_i \bigwedge_y R_t^{i,y} = R_{t+1}^{i,y})\right].$$

which reads "For all times t, if at time t, instruction 7 is being executed, then the program counter should advance to instruction 13 at time $t+1$ and the contents of all registers should remain unchanged." The translations for the other control flow instructions are very similar to the **jump** instruction. For the conditional jumps (**jpos, jzero, jneg**), we check that program counter is updated to the target line of the instruction in the case the condition holds. In the case of **halt** instruction, what we require is that the program counter is set to zero, and neither the program counter nor the contents of r_0 (the accumulator) changes after that point. The remaining instructions are self-explanatory, though note that in the case of pointer operations ($\uparrow j$) an additional quantification over all possible contents of that register is required, since at compile time the future contents of any register is unknown.

Arithmetic instructions The arithmetic instructions are more complicated, but follow simply from encoding a boolean circuit for performing the requisite operations. We first define the following boolean variables:

$R_t^{j,b}[i]$	At time t, i^{th} bit of register j is b
$C_t^b[i]$	At time t, i^{th} bit of the carry register is b

Figure 6: The indexed boolean variables

Note that the boolean variable $R_t^{j,y}$ now becomes a conjunction over the binary representation of y, and can be written as

$$\bigwedge_{1 \leq k \leq \lceil \log y \rceil} R_t^{j,y_k}[k]$$

where y_i is the i^{th} bit of y.

Having defined these variables, we can encode the arithmetic instructions, using the same ideas as above, as they are given in the Figure 7. To save space, the subformulas encoding the register consistency condition (all but register 0 remains unchanged) are omitted from the table.

2.3 Defining the Boolean Formulation

For a RAM program $\Pi = \{\pi_1, \ldots, \pi_k\}$ with running time bounded by the integer function $T(n)$, and input x, let $T = T(|x|)$. Define a function $\psi(\pi_j)$ which maps a RAM instructions π_j into its encoding given above. Quantifications of t, x, y are over $[1, \ldots, T]$ and of i, j are over $[1, \ldots, S]$. We first take the conjunction of all the clauses resulting from encoding the program.

Definition 1 *For a machine* Π *as given above, we define* $\Phi(\Pi) = \bigwedge_{i=1}^{k} \psi(\pi_i)$ *as the* satisfiability formulation *of* Π.

We will now do two things. We first replace variables which previously encoded integers with a set of variables corresponding to the individual bits of each integer, i.e., $R_t^{j,y}$ is replaced by the conjunction over $R_y^{j,y_k}[k]$'s. Secondly we transform our entire formula into 3CNF form. Consider a canonical list of the values of all the registers and the input. Assume we have m registers and m input integers. The list can be written $\{R_0, \ldots, R_m, I_0, \ldots I_m\}$. We define a new set of variables $\{X_i^t\}$ for $i = 0, 2m\log(S)$, $t = 0, \ldots, T$ corresponding to the values of each bit of each member of this list at each time step t. (Recall that we defined S as the space bound of the machine). By directly substituting these variables in for the previous variables, and making appropriate modifications to the equality tests, we arrive at a boolean formula. We then apply DeMorgan's Laws and convert this into 3CNF form.

Definition 2 *For the satisfiability formula* Φ *above, we define* Φ' *as the* boolean formulation *of* Π, $\Phi' = \bigwedge_{i=1}^{W} \tau_i$ *where* $\tau_i = (Y_{i_1}^{t_1} \vee Y_{i_2}^{t_2} \vee Y_{i_3}^{t_3})$, *where the literals* $Y_i^t = X_i^t$ *or* $\neg X_i^t$.

Theorem 2 $\Pi(x)$ *accepts iff* $\Phi'(\{X\}_i^t)$ *is satisfiable.*

Proof. The proof is analogous to Cook's Theorem. We have given a sentence which encodes the execution of the RAM, hence the satisfying assignments are in one-one correspondence with the accepting runs of the machine. ■

Note that although the RAM model we describe is deterministic, by modifying the jump instructions it is simple to implement nondeterminism. Moreover, our encoding never used the fact that the computation was deterministic. Hence, just as in Cook's reduction, our proof extends easily to nondeterministic RAM models.

Instruction	Boolean Formula
add j	$\bigwedge_t \bigwedge_{i,j} \bigwedge_{a_i,b_i,c_i} \Big[Q_t^n \wedge R_t^{0,a_i}[i] \wedge R_t^{j,b_i}[i] \wedge C_t^{c_i}[i] \rightarrow Q_{t+1}^{n+1} \wedge R_{t+1}^{0,a_i\oplus b_i\oplus c_i}[i] \wedge C_{t+1}^{(a_i\wedge b_i)\vee(a_i\wedge c_i)\vee(b_i\wedge c_i)}[i+1]\Big]$
add $=j$	$\bigwedge_t \bigwedge_i \bigwedge_{a_i,j_i,c_i} \Big[Q_t^n \wedge R_t^{0,a_i}[i] \wedge C_t^{c_i}[i] \rightarrow Q_{t+1}^{n+1} \wedge R_{t+1}^{0,a_i\oplus j_i\oplus c_i}[i] \wedge C_{t+1}^{(a_i\wedge j_i)\vee(a_i\wedge c_i)\vee(j_i\wedge c_i)}[i+1]\Big]$
add $\uparrow j$	$\bigwedge_t \bigwedge_{i,j,d} \bigwedge_{a_i,b_i,c_i} \Big[Q_t^n \wedge R_t^{0,a_i}[i] \wedge R_t^{j,d}[i] \wedge R_t^{d,b_i}[i] \wedge C_t^{c_i}[i] \rightarrow Q_{t+1}^{n+1} \wedge R_{t+1}^{0,a_i\oplus b_i\oplus c_i}[i] \wedge C_{t+1}^{(a_i\wedge b_i)\vee(a_i\wedge c_i)\vee(b_i\wedge c_i)}[i+1]\Big]$
sub j	$\bigwedge_t \bigwedge_{i,j} \bigwedge_{a_i,b_i,c_i} \Big[Q_t^n \wedge R_t^{0,a_i}[i] \wedge R_t^{j,b_i}[i] \wedge C_t^{c_i}[i] \rightarrow Q_{t+1}^{n+1} \wedge R_{t+1}^{0,a_i\oplus b_i\oplus c_i}[i] \wedge C_{t+1}^{(\overline{c_i}\wedge b_i)\vee(c_i\wedge(\overline{a_i}\vee b_i))}[i+1]\Big]$
sub $=j$	$\bigwedge_t \bigwedge_i \bigwedge_{a_i,j_i,c_i} \Big[Q_t^n \wedge R_t^{0,a_i}[i] \wedge C_t^{c_i}[i] \rightarrow Q_{t+1}^{n+1} \wedge R_{t+1}^{0,a_i\oplus j_i\oplus c_i}[i] \wedge C_{t+1}^{(\overline{c_i}\wedge b_i)\vee(c_i\wedge(\overline{a_i}\vee b_i))}[i+1]\Big]$
sub $\uparrow j$	$\bigwedge_t \bigwedge_{i,j,d} \bigwedge_{a_i,b_i,c_i} \Big[Q_t^n \wedge R_t^{0,a_i}[i] \wedge R_t^{j,d}[i] \wedge R_t^{d,b_i}[i] \wedge C_t^{c_i}[i] \rightarrow Q_{t+1}^{n+1} \wedge R_{t+1}^{0,a_i\oplus b_i\oplus c_i}[i] \wedge C_{t+1}^{(\overline{c_i}\wedge b_i)\vee(c_i\wedge(\overline{a_i}\vee b_i))}[i+1]\Big]$
half j	$\bigwedge_t \bigwedge_i \bigwedge_{a_i} \Big[Q_t^n \wedge R_t^{0,a_i}[i] \rightarrow Q_{t+1}^{n+1} \wedge R_{t+1}^{0,a_i}[i-1] \wedge \big(\bigwedge_y \bigwedge_{j\neq 0} R_t^{j,y} = R_{t+1}^{j,y}\big)\Big]$

Figure 7: Translations for arithmetic instructions

3 Completing the Protocol

All PCP protocols can now be revised to begin with a RAM description instead of a Turing Machine description, as shown in the previous section. For completeness we summarize in this section the remainder of one variant of the PCP protocol, at a high level, so that the reader may get a feel for the feasibility of implementing the protocol. This exposition is based primarily on the description of [AS98], which in turn depends on [BFL91] and [LFKN90]. Although [ALM+98] actually requires fewer queries than can be achieved in [BFL91], we employ the latter method because it is simpler and the total communication bandwidth and running times of the verifier are similar in both models.

Here we also explained why in our version of this protocol, the full 3CNF formula never needs to be explicitly written out by our verifier. It should be noted that the results of PCP protocols do not end here. Numerous beautiful and surprising optimizations have been derived in recent years. The results in this paper may be applied to any of these versions of PCP.

The problem of the input All PCP protocols require that the verifier and the prover both read the input string to the RAM M. It is for this reason that the verifier's running time is always at least linear, even though the time spent in verification is only logarithmic in M's running time. In our certification model, we are allowing M to read and accept a very long certificate. We explicitly assumed that V does not want to download this long certificate, but this certificate is the input to M, so V seemingly needs to take the time to read it.

It turns out that V does not have to read the entire certificate. Using a standard interpretation of the class *NP* we instead assume that the certificate is *guessed* by M nondeterministically. M then verifies that the certificate it guessed is correct. Using this, the only input to M is the program which we assume V already downloaded. It does not matter that the certificate is in fact provided by another source (namely, the prover.) If a certificate exists which M accepts, then the protocol will pass.

3.1 Arithmetization

Given a 3CNF formula Φ' over w variables $\vec{X}$, we describe a polynomial $P(\vec{x})$ which has a zero over $\{0,1\}^w$ if and only if Φ' is satisfiable. Let $\Phi' = \bigwedge \tau_i$. Then define a function A from boolean formulas to polynomials in the following way: $A(T) = 0$, $A(F) = 1$, $A(X) = (1-x)$, $A(\neg X) = x$, $A(Y_1 \vee Y_2 \vee Y_3) = A(Y_1)A(Y_2)A(Y_3)$.

Definition 3 *Let Φ' be a boolean formulation of Π as in Definition 2 above. Then the polynomial $P(\vec{x}) = \sum_i (A(\tau_i))^2$ is called the* boolean arithmetization *of the program Π.*

Lemma 1 *The boolean arithmetization $P(\vec{x})$ of Φ has a zero over $\{0,1\}^w$ iff Φ' is satisfiable.*

Proof. From the definition of P it is clear that there is an x such that $P(x) = 0$ has a zero at x only if each of its summands is 0. Each summand is a product, thus is zero is any member of the factors is zero. But these factors correspond to the arithmetization of literals, which have the value 0 if and only if the literal evaluates to true. ■

3.2 Compression

A general scheme is given in [BFL91] for converting sums of the above type into sums which can be efficiently zero tested. We briefly recount that transformation here. The goal is to produce an equivalent polynomial with more levels of nested sums, which as we will see below, allows more efficient zero testing.

Given a set of boolean variables $\{x_1, \ldots, x_w\}$ and a set $\{\tau_1, \ldots, \tau_m\}$ of disjunctions of 3 literals, we again let $\Phi = \bigwedge_{i=1}^m \tau_i$. Now instead of taking the conjunction of only the τ_i which appear in Φ, we want to conjunct over all possible disjunctions of length three. We then incorporate some additional logic to ignore the superfluous formulas which were introduced. First we need a decision procedure D which takes as input 3 variables $\{x_{i_1}, x_{i_2}, x_{i_3}\}$ and 3 bits $\{b_1, b_2, b_3\}$ to indicate whether the variables are negated. D accepts only if the resulting disjunction τ_i appears in Φ. In the general theory of this paper, $D(\vec{x}, \vec{b})$ could be compiled in RAM code and transformed into a boolean formulation, $\Delta(\vec{x}, \vec{b})$. Using this we can transform Φ into a new boolean formula, which we then arithmetize. The boolean formula inside the summation can be read as "either $(x \vee y \vee z)$ does not appear in Φ or else it satisfiable." Recall that A is a function which arithmetizes boolean formulas.

$$\begin{aligned}
A(\Phi) &= A(\bigwedge_{i=1}^{m} \tau_i) \\
&= \sum_{i=1}^{m} A(\tau_i) \\
&= \sum_{b_1=0}^{1}\sum_{b_2=0}^{1}\sum_{b_3=0}^{1}\sum_{i_1=1}^{n}\sum_{i_2=1}^{n}\sum_{i_3=1}^{n} A(\neg\Delta(b_1, b_2, b_3, x_{i_1}, x_{i_2}, x_{i_3}) \vee (b_1 \oplus x_{i_1}) \vee (b_2 \oplus x_{i_2}) \vee (b_3 \oplus x_{i_3}))
\end{aligned}$$

Note that this new formula in fact has zero exactly when Φ is satisfiable and hence it has a zero if and only if our original program Π accepts its input.

Definition 4 *We call the above transformation of a boolean formulation Φ, the* decision formulation *of Φ. We use this term indiscriminately to refer to both the logical and polynomial formulations.*

3.3 The Protocol

By applying 3.1 and 3.2 to a program Π, we find a polynomial F. Computing the value of $F(\vec{x})$ directly involves unwrapping the above sum and hence computing the summand $O(n^3)$ times. Since n is at least the running time of Π this computation is clearly too long. Yet the result of [LFKN90] shows that such a sum can be evaluated with the aid of a prover with an additive complexity increase instead of the normal multiplicative one. Hence by performing $O(3n)$ simple computations and queries to the prover, the verifier can be convinced that the formula is satisfiable. But $3n$ is still essentially the running time of Π. Another trick from BFL allows us to perform only $O(\log n)$ queries by quantifying not over the variables $\{x_1, \ldots, x_n\}$ themselves, but over the binary digits of their indices. Thus a conjunction such as

$$\sum_{i=1}^{n} \sum_{j=1}^{n} A\left(x_i \vee x_j\right)$$

is instead written as

$$\sum_{c_1=0}^{1} \cdots \sum_{c_{\log n}=0}^{1} \sum_{d_1=0}^{1} \cdots \sum_{d_{\log n}=0}^{1} A\left(x_{\sum_{i=1}^{\log n} c_i 2^i} \vee x_{\sum_{i=1}^{\log n} d_i 2^i}\right)$$

By applying this idea to our decision formulation of Φ we arrive at a sum which would require $O(\log n)$ steps of the LFKN protocol.

Definition 5 *The above transformation is called the* compressed formulation *of* Φ.

The final step The last step of the LFKN protocol requires the evaluation of the summand $\Delta(\vec{x}, \vec{b}) \vee \tau$ for one particular valuation of all the quantified boolean variables. The bottleneck in this computation is the evaluation of Δ. A simple way of performing the decision procedure of finding a given triple of literals in Φ would be to compute Φ completely and then search. To bypass this, recall that in the original boolean formulation, each variable name specifically referred to the line number of Π which gave rise to that variable. Moreover, within that line number, the precise time step and register values consider are also encoded in the variable name. Hence given any variable, determining if it appears in Φ reduces to determining if it appears in the encoding of a particular instruction π_i. This process can be encoded simply in logarithmic time, and hence the final step of the protocol does not add to our running time. Note also that it should now be clear why the verifier never needs to explicitly compute Φ. It is only ever needed to know how to query Φ and this can be done with a logarithmic time lookup procedure.

References

[ALM+98] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems, *J. of the ACM*, 45(3):501–555, 1998.

[AS98] S. Arora and S. Safra. Probabilistic checkable proofs: A new characterization of NP. *J. of the ACM*, 45(1):70–122, 1998.

[BFL91] L. Babai, L. Fortnow, and C. Lund. Non-deterministic exponential time has two-prover interactive protocols, *Computational Complexity*, pp. 3–40, 1991.

[BFLS90] L. Babai, L. Fortnow, C. Lund, and M. Szegedy. Checking computations in polylogarithmic time. *Proc. 31st Foundations of Computer Science*, pp. 16–25, 1990.

[BGKW88] M. Ben-Or, S. Goldwasser, J. Kilian, and A. Wigderson. Multi-prover interactive proofs: How to remove intractability assumptions. *Proc. 20th Symposium on Theory of Computing*, pp. 113–131, 1988.

[BK95] M. Blum and S. Kannan. Designing programs that check their work. *J. of the ACM*, 42(1):269–291, 1995.

[BLR93] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *J. of Computing and System Sciences*, 47(3):549–595, 1993.

[CMS99] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. Manuscript, 1999.

[CLSY90] J. Y. Cai, R. Lipton, R. Sedgewick, and A. Yao. Towards uncheatable benchmarks. *Proc. 8th Structure in Complexity Theory*, pp. 2–11, 1993

[DKL+] C. Dwork, S. Kannan, P. D. Lincoln, J. C. Mitchell, R. Rubinfeld, A. Scedrov. Interactive Proof-Carrying Code.

[DS92] C. Dwork and L. Stockmeyer. Finite state verifiers I: The power of interaction. *J. of the ACM*, 39(4):800–828, 1992.

[FGL+96] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, and M. Szegedy. Interactive proofs and the hardness of approximating cliques, *J. of the ACM*, 43(2):268–292, 1996.

[FN] U. Feige, K. Nissim. On the use of interactive proofs for formal program verification Manuscript, 1997

[FS88] L. Fortnow and M. Sipser. Interactive proof systems with a log space verifier. Manuscript, 1988.

[GLR+91] P. Gemmell, R. Lipton. R. Rubinfeld, M. Sudan, and A. Wigderson. Self-testing/correcting for polynomials and for approximate functions. *Proc. 23rd Symposium on Theory of Computing*, pp. 32–42, 1991.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. on Computing*, 18(1):186–208, 1989.

[GS86] S. Goldwasser and M. Sipser. Private coins versus public coins in interactive proof systems. *Proc. 18th Symposium on Theory of Computing*, pp. 59–68, 1986.

[Kozen] Kozen] D. Kozen Design and Analysis of Algorithms Springer-Verlag, 1994.

[Kil92] J. Kilian. A note on efficient zero-knowledge proofs and arguments. *Proc. 24th Symposium on Theory of Computing*, pp. 723–732, 1992.

[Kil94] J. Kilian. Improved efficient arguments (preliminary version). *Proc. Advances in Cryptology – CRYPTO*, Springer LNCS 963:311–324, 1995.

[KO97] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. *Proc. 38th Foundations of Computer Science*, pp. 364–373, 1997.

[Lip91] R. Lipton. New directions in testing. *Proc. DIMACS Workshop on Distr. Comp. and Cryptography*, pp. 191–202, 1991.

[LFKN90] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems, *J. of the ACM*, 39(4):859–868, 1992.

[Mer90] R. C. Merkle. A certified digital signature. *Proc. Advances in Cryptology – CRYPTO*, Springer LNCS 435:218–238, 1989.

[Mic94] S. Micali. CS proofs. *Proc. 35th Foundations of Computer Science*, pp. 436–453, 1994.

[NL96] G. C. Necula, P. Lee. Proof-carrying code. Technical Report CMU-CS-96-165, Computer Science Department, CMU, 1996.

[Papa] C. Papadimitriou.. Computational Complexity Addison Wesley, 1996

[PS] A. Polishchuk, D. A. Spielman. Nearly-linear Size Holographic Proofs. *Proc. 26th Symposium on Theory of Computing*, 1994.

[Sha90] A. Shamir. IP=PSPACE. *J. of the ACM*, 39(4):869–877, 1992.

[Spi96] D. A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Trans. on Information Theory*, 42(6):1723–1732, 1996.