Runtime Verification of Remotely Executed Code using Probabilistically Checkable Proof Systems

Tuğkan Batu Ronitt Rubinfeld Patrick White

November 15, 1999

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 machine 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

Probabilistically Checkable Proof Systems (PCPs) provide a surprising and elegant schema for efficient verification of (non)deterministic computations e.g. [BFL91, BFLS90, ALM+98, **?**]. By employing this mechanism a computation requiring T time steps can be probabilisitcally verified in $polylog(T)$ verification time steps. Yet the starting point of all current PCP constructions require a Turing Machine description of a language recognizer, and in practice interesting Turing Machines are rarely built. In this paper we show how to overcome this obstacle by reformulating the PCP theorems in terms of a random access machine (RAM). In doing so we revisit Cook's seminal reduction [**?**].

1.1 The Model

We assume a standard PCP model; namely a computationally bounded user ("Verifier" or "V") in interaction with an untrusted oracle ("Prover" or "P"). V would like to verify that P has computed that an input x is an element of a fixed language L. P generates a proof based on an arithmetization of the compuatation history of some Turing Machine M which accepts x.

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 RAM machines instead of Turing Machines. Although it is known that RAM machines and Turing machines are equivalent in power (cf. [Papa]), we show that a direct application of Cook's

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

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.

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 1. 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.

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	$\uparrow j$	$= j$	$r_0 \leftarrow x_j$	$r_0 \leftarrow x_{r_j}$	$r_0 \leftarrow j$
add	j	$\uparrow j$	$= j$	$r_0 \leftarrow r_0 + x_j$	$r_0 \leftarrow r_0 + x_{r_j}$	$r_0 \leftarrow r_0 + j$
sub	j	$\uparrow j$	$= j$	$r_0 \leftarrow r_0 - x_j$	$r_0 \leftarrow r_0 - x_{r_j}$	$r_0 \leftarrow r_0 - j$
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 1: Instruction set of a RAM

2.2 The Encoding

A correct computation of a RAM will be encoded as a satisfiable boolean formula. Each operand listed in Figure 1. is translated into a boolean sentence which will be quantified over all time steps of the execution of the 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 2: 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):

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

  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.

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

  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.

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

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.

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

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 machine 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.

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

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

which reads "For all times t, if at time t, instruction 7 is being executed, then the program counter should advance to instruction j 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.

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	$\bigwedge_t \left[Q_t^n \rightarrow \Big(\ (P_t \wedge Q_{t+1}^j) \vee (\neg P_t \wedge Q_{t+1}^{n+1})\Big) \wedge (\bigwedge_i \bigwedge_y R_t^{i,y} = R_{t+1}^{i,y})\right]$
jzero j	$\bigwedge_t \left[Q_t^n \rightarrow \Big(\ (Z_t \wedge Q_{t+1}^j) \vee (\neg Z_t \wedge Q_{t+1}^{n+1})\Big) \wedge (\bigwedge_i \bigwedge_y R_t^{i,y} = R_{t+1}^{i,y})\right]$
jneg j	$\bigwedge_t \left[Q_t^n \rightarrow \Big(\ (N_t \wedge Q_{t+1}^j) \vee (\neg N_t \wedge Q_{t+1}^{n+1})\Big) \wedge (\bigwedge_i \bigwedge_y R_t^{i,y} = R_{t+1}^{i,y})\right]$
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 3: Translations for move and control flow instructions

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

Figure 4: The indexed boolean variables

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:

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 5.

2.3 Defining the Boolean Formulation

For a RAM machine $\Pi = \{\pi_1, \ldots, \pi_k\}$ 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. 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 non-boolean 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 machine, 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 machine 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 5: Translations for arithmetic instructions

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

[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.

[DKL+] C. Dwork, S. Kannan, P. Lincoln, J. Mitchell, R. Rubinfeld, A. Scedrov. Untitled manuscript.

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

[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.