INFORMATION AND CONTROL 61, 159–173 (1984)

The Complexity of Promise Problems with Applications to Public-Key Cryptography*

SHIMON EVEN

Computer Science Department, Technion, Haifa, Israel

ALAN L. SELMAN

Computer Science Department, Iowa State University, Ames, Iowa 50011

AND

YACOV YACOBI

Electrical Engineering Department, Technion, Haifa, Israel

A "promise problem" is a formulation of a partial decision problem. Complexity issues about promise problems arise from considerations about cracking problems for public-key cryptosystems. Using a notion of Turing reducibility between promise problems, this paper disproves a conjecture made by Even and Yacobi (1980), that would imply nonexistence of public-key cryptosystems with NP-hard cracking problems. In its place a new conjecture is raised having the same consequence. In addition, the new conjecture implies that NP-complete sets cannot be accepted by Turing machines that have at most one accepting computation for each input word.

1. INTRODUCTION

This paper is concerned with several complexity issues about certain kinds of partial decision problems. The nature of these partial decision problems can be best explained by contrasting them with ordinary decision problems. A decision problem is given as a predicate $P(x)$. The question, of course, is

* This research was done while the second author visited the Computer Science Department, Technion, Haifa, Israel, with funds provided by the United States–Israel Educational Foundation (Fulbright Award), and while the third author visited the Electrical Engineering and Computer Science Department, University of California at San Diego, La Jolla, California. Some of the results of this paper were presented by the second and third authors at the 8th International Colloquium on Automata, Languages, and Programming, Aarhus, Denmark, July, 1982. This research was supported in part by the National Science Foundation under Grants MCS77–23493 A02 and MCS81–20263.

0019-9958/84 $3.00

to determine whether there exists an algorithm $\mathscr{A}$ that *solves* the problem, i.e., such that $\mathscr{A}(x)$ converges for all input instances x and such that

$$\forall x[\mathscr{A}(x) = \text{"yes"} \leftrightarrow P(x)].$$

In practice, one often encounters problems for which only a subclass of the domain of all instances is of concern. Such problems are here called *promise problems*. Informally, a promise problem has the structure

input x,
promise $Q(x)$,
property $R(x)$,

where Q and R are predicates. Formally, a promise problem is a pair of predicates (Q, R). The predicate Q is called the *promise*. A deterministic Turing machine M *solves* the promise problem (Q, R) if

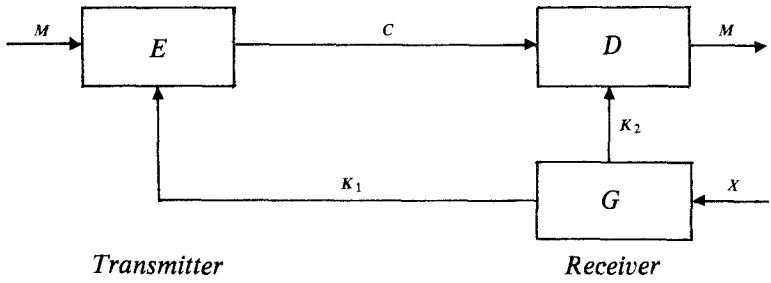
$$\forall x[Q(x) \rightarrow [M(x) \downarrow \wedge (M(x) = \text{"yes"} \leftrightarrow R(x))]].$$

(The notation $M(X) \downarrow$ means that M eventually halts on input x.). A promise problem (Q, R) is *solvable* if there exists a Turing machine M that solves it. If a Turing machine M solves (Q, R), then the language $L(M)$ accepted by M is a *solution* to (Q, R).

The study of problems with this format is certainly not new. For partial recursive functions one wants a program that computes correctly on its domain. And, techniques for establishing correctness of a program are typically distinct from halting issues. Problems of this kind have also arisen in context-free language theory (Ullian, 1967).

Complexity issues about promise problems arise from Even and Yacobi's (1980) work in public-key cryptography. Since their work is the primary motivation for the complexity results that follow, and since we wish to draw conclusions about public-key cryptosystems, the model of a public-key cryptosystem which they use is now given.

A public-key cryptosystem consists of three fixed and publicly known deterministic algorithms E, D, and G that operate in polynomial time. The diagram gives the basic layout; E is the *encryption algorithm*, D is the *decryption algorithm*, and G is the *key generator*; M, C, K_1, K_2, and X will be binary words, called the *message*, *cryptogram*, *encryption key*, *decryption key*, *and trap-door*, respectively. Prior to transmission of messages M of length n, the receiver generates X, say randomly, where $|X|$ is a polynomial in n, and then uses X to compute the pair $(K_1, K_2) = G(X)$. K_1 is made publicly known, but X and K_2 remain private. The encryption–decryption key pair may be used for encoding and decoding purposes for a relatively long time.



When a transmitter wants to send a message M of length n to a receiver, he computes $C = E(K_1, M)$ and sends C on an open channel. The receiver, knowing K_2, reconstructs M by $M = D(K_2, C)$. It is assumed that, for every X, if $(K_1, K_2) = G(X)$, then $M = D(K_2, E(K_1, M))$. To the extent that n is a parameter, this system is similar to Brassard's transient key cryptosystem (Brassard, 1983).

The inverse condition guarantees that the function $\lambda ME(K_1, M)$ is one–one, for each K_1 that is the public-key generated by G for some X. $\lambda ME(K_1, M)$ may very well not be onto.

A basic issue is whether there exist public-key cryptosystems with hard cracking problems. The cracking problem is the problem of computing M such that $E(K_1, M) = C$ *if such* M *exists*. Therefore, the cracking problem for a fixed public-key cryptosystem is describable as a promise problem in the following way.

input n, K_1, C and M' (where $|M'| = n$),
promise there exists X such that K_1 is the public-key generated by G on input X and there exists a message M, $|M| = n$, such that $E(K_1, M) = C$,
property $M' \geqslant M$, where M is the message which satisfies $E(K_1, M) = C$.

Since $\lambda ME(K_1, M)$ is one–one, there is at most one M which satisfies $E(K_1, M) = C$. Thus, if the promise is true, the question of whether the numerical value of M' is greater or equal to the numerical value of M is always meaningful, and has a positive or negative answer. Furthermore, the computational version of the cracking problem (find M) is polynomial-time equivalent to the version given here. Given M such that $E(K_1, M) = C$, simply check whether $M' \geqslant M$. Conversely, if the promise holds for n, K_1, and C, an algorithm $\mathscr{A}$ that solves the above formulation of the cracking problem can be used in a binary search to find M, and this computation runs in polynomial time relative to $\mathscr{A}$.

A public-key cryptosystem can be deemed secure only if every algorithm

that solves the promise problem formulation of its cracking problem is inefficient. Attempts to formulate the cracking problem as an ordinary decision problem would lead to faulty considerations. To see this, let Q and R denote the promise and property predicates, respectively, of the cracking promise problem and, tentatively, define the conjunction $Q \cap R$ to be the cracking decision problem. Now, one might have a public-key cryptosystem, say $\mathcal{S}$, such that the cracking decision problem $Q \cap R$ is difficult to solve. But if there is an efficient algorithm that solves the cracking promise problem, i.e., that efficiently finds M when the promise predicate Q is true and that gives "garbage" output when Q is false, then $\mathcal{S}$ is not a usable system. On the other hand, if the cracking promise problem (Q, R) has no efficient solution, then the cracking decision problem $Q \cap R$ is difficult to solve also, because any algorithm that solves $Q \cap R$ also solves (Q, R).

This paper is organized so that Sections 2 and 3 provide basic complexity notions and results about promise prolems. Section 4 shows that a conjecture raised by Even and Yacobi (1980) is false. That conjecture would, if it were true, imply that public-key cryptosystems with NP-hard cracking problems do not exist. In Section 5 a new conjecture is raised that has the same consequence. In addition, the new conjecture implies that NP-complete sets cannot be accepted by Turing machines that have at most one accepting computation for each input word.

NP-hardness is a worst case notion, and, of course, one needs to know that the cracking problem of a given public-key cryptosystem is hard for almost all cases. If a cryptosystem for which the cracking problem is hard does not exist when the worst case approach is taken, then certainly it does not exist when an average or most-cases approach is used.

2. Complexity Concepts for Promise Problems

Recall that if M is a deterministic Turing machine that solves a promise problem (Q, R), then the language $L(M)$ accepted by M is called a *solution* to (Q, R). By means of this technical device every promise problem specifies a class of languages and solutions to promise problems can be described and manipulated set theoretically. We require notation for promise problems that have solutions in NP. One possibility is to extend the definition of NP to include promise problems. However, in order to keep NP sacrosanct for decision problems (encoded as languages), the following notation is introduced instead.

Definition 1. NPP is the class of all promise problems (Q, R) such that (Q, R) has a solution in NP. Co-NPP is the class of all promise problems (Q, R) such that $(Q, \sim R)$ is in NPP.

A promise problem (Q, R) belongs to co-NPP if and only if (Q, R) has a solution in co-NP. Also note that a recursive language L is a solution to (Q, R) if and only if $\sim L$ is a solution to $(Q, \sim R)$. Thus, L is a solution in NP to (Q, R) if and only if $\sim L$ is a solution in co-NP to $(Q, \sim R)$.

Every set S in NP may be considered to be a promise problem (Σ^*, S) with the trivial promise Σ^*, where S is a language over the finite alphabet Σ. (Occasionally we use sets instead of predicates when we name a promise problem.) If S is in NP (co-NP, NP $\cap$ co-NP), then (Σ^*, S) is in NPP (co-NPP, NPP $\cap$ co-NPP, respectively). In this way, NPP is a proper extension of NP, co-NPP is a proper extension of co-NP, and NPP $\cap$ co-NPP is a proper extension of NP $\cap$ co-NP.

If a promise problem (Q, R) is in NPP $\cap$ co-NPP, then by definition there is a solution in NP and there is a solution in co-NP. However, there is no reason to believe that (Q, R) has a solution in NP $\cap$ co-NP. This simple observation suggests significant structural differences between the classes NPP $\cap$ co-NPP and NP $\cap$ co-NP, and later results will bear this out.

Let $\leqslant_m^P$ and $\leqslant_T^P$ denote polynomial-time many–one and Turing reducibilities, respectively. (This notation was established in Ladner, Lynch, and Selman (1975).) In accordance with generally accepted usage, recall that a language L is NP-complete if L is in NP and every set in NP is $\leqslant_m^P$-reducible to L and recall that a language L is NP-hard if every set in NP is $\leqslant_T^P$-reducible to L. (Garey and Johnson (1979) contains a useful discussion and terminological history which explains why NP-complete and NP-hard are defined by different reducibilities.)

DEFINITION 2. A promise problem (Q, R) is NP-*hard* if it is solvable and every solution L of (Q, R) is NP-hard.

It follows from the definition that if an NP-hard promise problem has a tractable solution (i.e., in P), then $P = \text{NP}$. In particular, if a public-key cryptosystem has an NP-hard cracking problem, then the system can be cracked (in worst case) in polynomial time only if $P = \text{NP}$.

For an oracle Turing machine M with oracle set A, let $L(M, A)$ denote the language accepted by M with oracle A. According to Definition 2, (Q, R) is NP-hard if and only if, for every set S in NP and for every solution A of (Q, R), there is an oracle Turing machine M that operates in polynomial time so that $S = L(M, A)$.

DEFINITION 3. A promise problem (Q, R) is *uniformly* NP-hard if it is solvable and for every set S in NP, there is an oracle Turing machine M that operates in polynomial time such that, for all solutions A of (Q, R), $S = L(M, A)$.

Uniformly NP-hard obviously implies NP-hard. The converse can be

expected to be false but no proof is yet known. In any case, the main results of the next section will hold for both notions.

The concepts thus far defined can now be used for definitions of reductions and uniform reductions between promise problems.

DEFINITION 4. A promise problem (Q, R) is *Turing reducible in polynomial time* to a promise problem (S, T), in symbols, $(Q, R) \leqslant_T^{PP} (S, T)$, if, for every solution A of (S, T), there is an oracle Turing machine M that operates in polynomial time such that M with oracle A solves (Q, R).

DEFINITION 5. A promise problem (Q, R) is *uniformly Turing reducible in polynomial time* to a promise problem (S, T), $(Q, R) \leqslant_{UT}^{PP} (S, T)$, if there is an oracle Turing machine M that operates in polynomial time such that, for every solution A of (S, T), M with oracle A solves (Q, R).

LEMMA 1. (i) $\leqslant_T^{PP}$ *and* $\leqslant_{UT}^{PP}$ *are transitive relations.*

(ii) $(Q, R) \leqslant_{UT}^{PP} (S, T)$ *implies* $(Q, R) \leqslant_T^{PP} (S, T)$.

(iii) *A solvable promise problem* (Q, R) *is* NP*-hard if and only if, for every set* S *in* NP, $(\sum^*, S) \leqslant_T^{PP} (Q, R)$.

(iv) *A solvable promise problem* (Q, R) *is uniformly* NP*-hard if and only if, for every set* S *in* NP, $(\sum^*, S) \leqslant_{UT}^{PP} (Q, R)$.

Whether $\leqslant_T^{PP}$ implies $\leqslant_{UT}^{PP}$ is an open question.

3. ELEMENTARY RESULTS

Since we are focusing on polynomial time complexity issues, let us assume henceforth that, for all promise problems (Q, R) mentioned, both Q and R are recursive predicates. Furthermore, we assume that if M is a Turing machine that solves (Q, R) then M halts on every input. Therefore, every solution to a promise problem is a recursive set. The solution criterion becomes

$$Q(x) \rightarrow (M(x) = \text{"yes"} \leftrightarrow R(x)).$$

The solutions to a promise problem (Q, R) can be completely characterized set theoretically: A recursive set A is a solution to (Q, R) if and only if $A = (Q \cap R) \cup B$, where $Q \cap B = \varnothing$ and B is recursive. In particular, every promise problem is solvable; $Q \cap R$, R, and $R \cup \sim Q$ are solutions; $Q \cap R$ is the smallest solution, and $R \cup \sim Q$ is the largest solution. If (Q, R) is an NP-hard promise problem, then $Q \cap R$, R, and $R \cup \sim Q$ are NP-hard sets.

Promise problems with tractable promise can be analyzed rather completely. First of all, R is NP-hard, and Q is in P does not imply that (Q, R) is NP-hard. (Take Q to be empty or finite and observe that $Q \cap R$ is either empty or finite and so, assuming $P \neq NP$, (Q, R) is not NP-hard.) To obtain a nontrivial example, use Ladner's result (Ladner, 1975) to obtain an NP-complete set R and a set Q in P such that $Q \cap R$ is not NP-hard (although $Q \cap R$ is in NP-P assuming $P \neq$ NP). Thus, (Q, R) is not NP-hard.

THEOREM 1. *If R is* NP*-hard, Q is in P, and (Q, R) has a solution in P, then $(\sim Q, R)$ is* NP*-hard.*

Proof. Let R be NP-hard, Q in P, and let A in P be a solution to (Q, R). Let B be an arbitrary solution to $(\sim Q, R)$. To show that $(\sim Q, R)$ is NP-hard, it suffices to show $R \leqslant_T^P B$. This is accomplished by the following algorithm with oracle set B.

input x;
if $Q(x)$
  then if $x \in A$ $\{Q(x) \rightarrow (x \in A \leftrightarrow R(x))\}$
      then accept
      else reject
  else if $x \in B$ $\{\sim Q(x) \rightarrow (x \in B \leftrightarrow R(x))\}$
      then accept
      else reject.

LEMMA 2. *If Q is in P and A is a solution to (Q, R), then $Q \cap R \leqslant_{UT}^{PP} A$.*

Proof. Let Q belong to P and let A be any solution to (Q, R). The reduction follows from the obsevation that $Q \cap R = Q \cap A$.

As an immediate consequence we have

THEOREM 2. *If Q is in P, then (Q, R) is an* NP*-hard promise problem if and only if $Q \cap R$ is an* NP*-hard set.*

Theorem 2 can be used to generate many interesting examples of NP-hard promise problems in NPP. The technique is this: Let R be any known NP-complete problem and let $Q \cap R$ be a refinement that is still NP-complete, where Q belongs to P. Then, (Q, R) is NP-hard. For example, let SAT be an encoding of the satisfactory formulas of propositional logic, and let **3** be an encoding of all formulas with three literals per clause. **3** $\cap$ SAT is the well-known NP-complete set 3SAT. Since **3** is in P, Theorem 2 applies. Hence, (**3**, SAT) is an NP-hard promise problem in NPP.

THEOREM 3. *If Q belongs to P and (Q, R) is an* NP*-hard problem, then (Q, R) is uniformly* NP*-hard.*

Proof. Since (Q, R) is NP-hard, $Q \cap R$ is an NP-hard solution. Let M be an oracle Turing machine that operates in polynomial time and that, by use of Lemma 2, uniformly reduces $Q \cap R$ to solutions of (Q, R). Let S be any set in NP. Then, the machine which $\leqslant_T^P$-reduces S to $Q \cap R$ followed by M uniformly reduces S to each solution of (Q, R). ∎

For promises in NP, we have

LEMMA 3. *If Q is in* NP, *then (Q, R) is in* NPP *if and only if $Q \cap R$ is in* NP.

Proof. The proof from right to left follows directly from the definition of NPP. For the proof in the other direction, let A be a solution in NP to (Q, R). Thus, $Q \cap A$ is in NP, but $Q \cap A = Q \cap R$. ∎

4. On the Class NPP $\cap$ *co*-NPP

Let us consider the cracking problem for public-key cryptosystems once again, and let us note that cracking problems are in NPP $\cap$ *co*-NPP. To see that the cracking problem has a solution in NP, the procedure on input n, K_1, C, and M' is to guess X and M, test whether X and M satisfy the promise and if so, then accept if $M' \geqslant M$. Similarly, it is easy to see that the cracking problem is in *co*-NPP. (In this case, accept if $M' < M$.)

It is well known that NP $\cap$ *co*-NP contains an NP-hard set if and only if NP is closed under complements (cf. Brassard, 1979 or Selman, 1974). Hence, the former property is unlikely to be true. It is hypothesized in Even and Yacobi (1980) that there exist no NP-hard promise problems in NPP $\cap$ *co*-NPP. By the remarks in the preceding paragraph, a consequence of this hypothesis is nonexistence of public-key cryptosystems with NP-hard cracking problems. By the following lemma it is certainly reasonable to conjecture that NPP $\neq$ *co*-NPP.

LEMMA 4. NPP = *co*-NPP *if and only if* NP = *co*-NP.

Proof. Since NPP and co-NPP are extensions of NP and *co*-NP, respectively, the proof from left to right is trivial. The converse implication follows from the observation that if A is a solution to (Q, R), then $\bar{A}$ is a solution to $(Q, \sim R)$. ∎

However, we now provide an example of an NP-hard promise problem in NPP $\cap$ *co*-NPP. Let $\oplus$ denote the logical operator "exclusive or." Let SAT denote the NP-complete satisfiability problem. We will take the liberty also of writing SAT as a predicate, so that SAT(x) asserts that x is satisfiable. Let EX denote the predicate defined by $\mathrm{EX}(x, y) \leftrightarrow \mathrm{SAT}(x) \oplus \mathrm{SAT}(y)$. Define SAT $1 = \lambda x \lambda y \mathrm{SAT}(x)$, so that $\mathrm{SAT1}(x, y) \leftrightarrow \mathrm{SAT}(x)$.

THEOREM 4. (i) $(EX, SAT1) \in \text{NPP} \cap co\text{-NPP}$.

(ii) $(EX, SAT1)$ *is* NP-*hard.*

Proof. (i) Let x and y be input words and suppose the promise $\text{EX}(x, y)$ is true. Then, $\sim\text{SAT}(x)$ is equivalent to the predicate $\text{SAT}(y)$. Thus, it is evident that both (EX, SAT1) and (EX, $\sim$SAT1) belong to NPP.

(ii) Let A be any solution to (EX, SAT1). (Technically, A is a language consisting of encoded ordered pairs; we will write $\langle x, y\rangle \in A$ to denote membership in A.) If $\text{SAT}(x) \oplus \text{SAT}(y)$, then $\langle x, y\rangle \in A \leftrightarrow \text{SAT}(x)$. To show that A is NP-hard, it suffices to show that $\text{SAT} \leqslant_T^P A$, by the following iterative algorithm with oracle A.

Let ψ be a program variable that ranges over propositional formulas. Let $\phi(\sigma_1,..., \sigma_n)$ be an input formula with Boolean variables $\sigma_1,..., \sigma_n$.

$\psi := \phi(\sigma_1,..., \sigma_n)$;
for $i := 1$ **to** n **do**
  {ψ has free variables $\sigma_i,..., \sigma_n$}
  if $\langle \psi(0, \sigma_{i+1},..., \sigma_n), \psi(1, \sigma_{i+1},..., \sigma_n)\rangle \in A$
    then $\psi := \psi(0, \sigma_{i+1},..., \sigma_n)$
    else $\psi := \psi(1, \sigma_{i+1},..., \sigma_n)$;
{ψ is a variable-free Boolean expression}
if ψ has value 1
  then accept {ϕ is satisfiable}
  else reject {ϕ is not satisfiable}.

The algorithm clearly operates in polynomial time. If the accept state is reached, then a satisfying assignment for ϕ has been found. Conversely, suppose ϕ is satisfiable. We claim that the loop preserves satisfiability of ψ. At each execution of the loop body, if $\psi(\sigma_1,..., \sigma_n)$ is satisfiable, then $\psi(0, \sigma_{i+1},..., \sigma_n)$ is satisfiable or $\psi(1, \sigma_{i+1},..., \sigma_n)$ is satisfiable. If exactly one of these is satisfiable, then the promise is true and the oracle query provides correct information. If both of these are true, then ψ remains true independent of the value of the oracle query. Hence, the algorithm correctly reduces SAT to the solution A of (EX, SAT1) in polynomial time. ∎

Since the algorithm just given does not depend on choice of solution, we have the following corollary.

COROLLARY 1. (EX, SAT1) *is uniformly* NP-*hard.*

COROLLARY 2. *For each promise problem* (Q, R) *in* NPP, $(Q, R) \leqslant_T^{PP}$ (EX, SAT1).

Proof. Let $(Q, R) \in \text{NPP}$. Let L be a solution in NP to (Q, R). Let A be an arbitrary solution to (EX, SAT1). Let M be an oracle Turing machine

that operates in polynomial time such that $L = L(M, A)$. By definition, M with oracle A solves (Q, R). ∎

Whereas NP ∩ *co*-NP probably does not contain a set that is complete for NP ∩ *co*-NP (Sipser, 1982), we see here that NPP ∩ *co*-NPP does contain the complete promise problem (EX, SAT1). The results of this secton indicate that complexity classes of promise problems have different structural properties than do complexity classes of ordinary decision problems.

In anticipation of the issues to be raised in the next section, it is worth noting the complexity of the promise predicate EX. Following Papadimitriou and Yannakakis (1982), define $D^P = \{L_1 \cap L_2 | L_1 \in \text{NP} \text{ and } L_2 \in co\text{-NP}\}$. Trivially, $\text{NP} \subseteq D^P$, $\text{co-NP} \subseteq D^P$, and $D^P \subseteq \Delta_2^P$ (where Δ_2^P is defined to be P^{NP}). It is expected that each of these inclusions is proper.

PROPOSITION 1. EX *is complete for* D^P.

Proof. Let x and y be propositional formulas and assume without loss of generality that x and y have no propositional variables in common. Then $\text{EX}(x, y) \leftrightarrow \text{SAT}(x \vee y)$ and $\sim\text{SAT}(x \wedge y)$, and so EX belongs to D^P. Let SAT-UNSAT be the decision problem defined by

$$\langle x, y \rangle \in \text{SAT-UNSAT} \leftrightarrow \text{SAT}(x) \text{ and } \sim\text{SAT}(y).$$

It is shown in Papadimitriou and Yannakakis (1982) that SAT-UNSAT is complete for D^P. Finally, a straightforward reduction of SAT-UNSAT to EX is given by

$$\langle x, y \rangle \in \text{SAT-UNSAT} \leftrightarrow \text{EX}(x, 0) \text{ and } \text{EX}(1, y). \quad ∎$$

5. A New Conjecture

We have just seen that there does exist an NP-hard promise problem (Q, R), in NPP ∩ *co*-NPP, but with $Q \in D^P$ and probably not in NP. We conjecture that promise problems with the first two properties cannot have Q in NP. More precisely,

Conjecture. There exists no promise problem (Q, R) such that

(i) $(Q, R) \in \text{NPP} \cap co\text{-NPP}$,

(ii) (Q, R) is NP-hard, and

(iii) Q is in NP.

PROPOSITION 2. *The conjecture implies that public-key cryptosystems with* NP*-hard cracking problems do not exist.*

To see the correctness of this proposition, simply observe (cf. the Introduction) that the promise predicate for cracking problems is in NP. This conjecture has even larger interest. We consider the set $\mathscr{U}$ of problems in NP that have unique solution (Valiant, 1976). That is, $\mathscr{U}$ is the set of all languages L for which there is a nondeterministic Turing machine M that witnesses $L \in \mathrm{NP}$ such that for every input x to M, M has at most one accepting computation. Whether $\mathscr{U} = \mathrm{NP}$ is a well-known open problem. (Aspects of this problem are discussed in Book, Long and Selman (1982), Geske and Grollmann (1983), and Rackoff (1982).)

THEOREM 5. *The conjecture implies* $\mathrm{NP} \neq \mathscr{U}$.

Proof. We will prove the contrapositive. Assume $\mathrm{NP} = \mathscr{U}$ and let L be any NP-complete set contained in $\mathscr{U}$. A promise problem (Q, R) is to be constructed that satisfies the conditions of the conjecture. Let M be a nondeterministic Turing machine that is a witness to $L \in \mathscr{U}$ and that operates in polynomial time p. Assume without loss of generality that M can make at most two distinct next moves in any configuration. Then, every computation of M on an input word x corresponds in a natural way to a binary choice sequence y of length at most $p(|x|)$. Let $\leqslant$ be any standard polynomial time computable ordering of the binary strings. Define the predicates Q and R by

$$Q(x, z) \leftrightarrow x \in L$$

and

$$R(x, z) \leftrightarrow x \in L \text{ and } z \leqslant y,$$

where y is the unique binary choice sequence that causes M to accept x.

By definition, $Q \in \mathrm{NP}$. It is easy to see that $(Q, R) \in \mathrm{NPP}$. Namely, a solution to (Q, R) that belongs to NP is given by the following nondeterministic procedure: Given x and z, guess a choice sequence y whose length is within the bound $p(|x|)$ and check to determine whether y causes M to accept x. If so, and if $z \leqslant y$, then accept. Otherwise, do not accept.

Similarly, it is easy to see that $(Q, \sim R)$ belongs to NPP. Therefore, (Q, R) belongs to $\mathrm{NPP} \cap co\text{-}\mathrm{NPP}$.

We now use the premise that L is NP-complete in order to show that (Q, R) is NP-hard. Let A be any solution of (Q, R). To show that A is NP-hard, it suffices to show that $L \leqslant_T^P A$, and this is accomplished by the following algorithm with oracle A:

(1) input x;

(2) apply a binary search to the set of strings of length $\leqslant p(|x|)$ to try to find the largest string z such that $\langle x, y \rangle \in A$;

(3) **if** step 2 does not output a string z
(4) **then** reject
(5) **else if** z is an accepting choice sequence of M for x
(6) **then** accept
(7) **else** reject.

Since step (2) can be performed in time $O(\log(2^{p(|x|)})) = O(p(|x|))$, the algorithm runs in polynomial time. We argue that the algorithm correctly reduces L to A. Suppose $x \in L$. Then, the promise $Q(x, z)$ is satisfied. Hence, $\langle x, z\rangle \in A \leftrightarrow R(x, z)$. The predicate R enjoys the property

$$R(x, z_1) \wedge (z_2 \leqslant z_1) \rightarrow R(x, z_2).$$

Therefore, step (2) does find a largest string z such that $\langle x, z\rangle \in A$. By definition of R, this string z must be the unique accepting choice sequence of M on input x. Hence, the algorithm accepts x.

Conversely, suppose the algorithm accepts input x. Then step (5) is reached. Hence, $x \in L(M)$. ∎

It may be worth noting that $\mathscr{U}$ is closed under $\leqslant_m^P$-reductions. Therefore, $NP = \mathscr{U}$ if and only if $\mathscr{U}$ contains an NP-complete set, and so we have

COROLLARY 3. *If the conjecture is true, then no* NP*-complete set can be accepted by any Turing machine that has at most one accepting computation for each input word.*

A variant of the conjecture is indeed true, assuming NP $\neq$ *co*-NP. This is shown in the next and final theorem. Unfortunately, the variant does not seem to have the same nice consequences for public-key cryptosystems.

THEOREM 6. NP = *co*-NP *if and only if there exists a promise problem* (Q, R) *such that*

(i) $(Q, R) \in$ NPP $\cap$ *co*-NPP,
(ii) (Q, R) *is uniformly NP-hard,*
(iii) Q *is in co*-NP.

The proof will require some facts about oracle Turing machines. First, an oracle Turing machine is a multitape Turing machine with a distinguished work tape, the *query tape*, and three distinguished states QUERY, YES, and NO. At some step of a computation on an input string x, M may transfer into the state QUERY. In state QUERY, M transfers into the state YES if the string currently appearing on the query tape is in some *oracle set* A; otherwise, M transfers into the state NO; in either case the tape is instantly erased.

Every oracle Turing machine M_1 is equivalent to an oracle Turing machine M_2 such that for every input string x and every query string w, M_2 on input x never enters state QUERY with w written on its query tape more than once. Moreover, if M_1 operates in polynomial time, then so does M_2. Now we will describe M_2. M_2 will simulate M_1, but as it does so will build two finite "tables" T_Y and T_N. At all times, $T_Y \cap T_N = \varnothing$. T_Y will contain each query string that receives a YES answer from its oracle and T_N will contain each query string that receives a NO answer from its oracle. One of the work tapes of M_2 is used to store (an appropriate encoding of) T_Y, another of the work tapes M_2 is used to store (an appropriate encoding of) T_N, and both these tapes are initially empty. On input x, M_2 begins a simulation of M_1. Whenever this simulated computation is to enter a query configuration with a word w written on the query tape, M_2 first determines whether the string w belongs to $T_Y \cup T_N$ already. If $w \in T_Y \cup T_N$, then M_2 erases its query tape and continues its simulation of M_1 in state YES if $w \in T_Y$ and in state NO if $w \in T_N$. (In particular M_2 does not enter its query state.) If $w \notin T_Y \cup T_N$, then M_2 enters state QUERY and writes the string w onto T_Y if the transfer state is YES, and onto T_N otherwise. Clearly, M_2 is equivalent to M_1 and M_2 operates in polynomial time if M_1 does. Given an input word x and oracle set A, M_2 on input x never queries A more than once about any string w, and T_Y and T_N are maintained to insure consistency of oracle responses.

Now the proof of Theorem 6 is given.

Proof. Suppose NP = *co*-NP and let L be any NP-complete set. Then, (Σ^*, L) satisfies all the conditions of the theorem.

To prove the converse, let L belong to NP and let M be a deterministic oracle Turing machine that uniformly reduces L to solutions of (Q, R). Then, M with its accepting and rejecting states reversed (call this machine M') reduces $\sim L$ to solutions of (Q, R). In accordance with the argument just given, M' on an input x builds tables T_Y and T_N and so does not query its oracle about any word w more than once. Let M_i, $i = 1, 2$, be NP-acceptors that solve (Q, R) and $(Q, \sim R)$, respectively, and let M_3 be an NP-acceptor for $\sim Q$. We now describe an NP-acceptor N for $\sim L$. On input x, N begins a simulation of M' on x but replaces each query w to the oracle by simulations of w on the NP-acceptors M_i, $i = 1, 2, 3$, according to the following nondeterministic algorithm. Machine N builds tables T_Y and T_N also and initially $T_Y = T_N = \varnothing$.

if $w \in T_Y$
  then continue simulation of M' in the YES state;
if $w \in T_N$
  then continue simulation of M' in the NO state;

if $w \notin T_Y \cup T_N$
  then if M_1 accepts w
    $\rightarrow$ **begin**
      $T_Y := T_Y \cup [w]$;
      continue simulation of M' in the YES state
      end
  $\square$ (M_2 accepts w) or (M_3 accepts w)
    $\rightarrow$ **begin**
      $T_N := T_N \cup [w]$;
      continue simulation of M' in the NO state
      end.

If the algorithm is executed on a word w and the guard that is chosen does not evaluate to true, then the simulation by N of M' on input x is to terminate without accepting. For every input word x to N, the simulation by N of M' on x can be completed, i.e., there is a computation of N on x that reaches one of the accepting or rejecting states of M'. To see this, observe that $w \in Q$ implies M_1 accepts w if and only if $w \in R$ and M_2 accepts w if and only if $w \in \sim R$, and $w \in \sim Q$ implies M_3 accepts w. Thus, for each word w that is not already in $T_Y \cup T_N$, at least one of the guards can evaluate to true.

It is obvious that the language accepted by N belongs to NP. We need to see that $\sim L$ is this language. First we will show that if x is accepted by N, then $x \in \sim L$. Consider a fixed accepting computation of N on x and consider the final values of the tables T_Y and T_N that N constructs. Since N simulates M', M' must accept x with any oracle A such that $T_Y \subseteq A$ and $T_N \subseteq \sim A$. Since M' uniformly reduces $\sim L$ to solutions of (Q, R), if a solution A of (Q, R) can be found such that $T_Y \subseteq A$ and $T_N \subseteq \sim A$, then $x \in \sim L$ follows from the previous statement. We claim that the set $A = L(M_1) - T_N$ is a solution of (Q, R), that $T_Y \subseteq A$ and that $T_N \subseteq \sim A$. If $w \in Q$, then $w \in A$ if and only if $w \in R$, because $L(M_1)$ is a solution of (Q, R), and so A is a solution of (Q, R) also. Obviously, $T_N \subseteq \sim A$. $T_Y \subseteq A$ because the algorithm places a word w into T_Y only if M_1 accepts w and because $T_Y \cap T_N = \varnothing$. This completes the proof in one direction.

Now let $x \in \sim L$ and let us see that N has an accepting computation on input x. Consider the final values of T_Y and T_N that are constructed by M' when M' is executed on input x with $L(M_1)$ as the oracle. Recall that this computation accepts x, since $L(M_1)$ is a solution of (Q, R). We claim there is a computation of N on input x for which the final values of its table are also T_Y and T_N. It follows immediately that this computation accepts x. To establish our claim, first note that $T_Y \subseteq L(M_1)$ and $T_N \subseteq \sim L(M_1)$. If $w \in T_Y$, it follows that the guard "M_1 accepts w" is true and therefore this computation of the algorithm places w into its table of YES responses. If

$w \in T_N$, then $w \notin L(M_1)$ and so either $w \in Q \cap \sim R$ or $w \in \sim Q$. Thus, the guard "(M_2 accepts w) or (M_3 accepts w)" is true. This computation of the algorithm places w into its table of NO responses. This completes the proof of the claim. Hence, if $x \in \sim L$, then N accepts x.

We proved that $\sim L$ is the language accepted by N; therefore, $\sim L \in$ NP. Since L is an arbitrary language in NP, NP = *co*-NP follows. ∎

RECEIVED: June 13, 1983; ACCEPTED: January 26, 1984

REFERENCES

BOOK, R., LONG, T., AND SELMAN, A. (1982) Qualitative controlled relativizations of complexity classes, manuscript.

BRASSARD, G. (1979), A note on the complexity of cryptography, *IEEE Trans. Inform. Theory* **IT-25**, No. 2, 232–233.

BRASSARD, G. (1983), Relativized cryptography, *IEEE Trans. Inform. Theory*, **IT-29**, No. 6, 877–894.

EVEN, S., AND YACOBI, Y. (1980), Cryptography and NP-completeness, "Proc. 7th Colloq. Automata, Lang. Programming," Lecture Notes in Computer Science Vol. 85, pp. 195–207, Springer-Verlag, Berlin/New York.

GAREY, M., AND JOHNSON, D. (1979), "Computers and Intractability: A Guide to the Theory of NP-Completeness," Freeman, San Francisco.

GESKE, J. AND GROLLMANN, J. (1983), Relativizations of unambiguous and random polynomial time classes, manuscript.

LADNER, R. (1975), On the structure of polynomial time reducibility, *J. Assoc. Comput. Mach.* **22**, 155–171.

LADNER, R., LYNCH, N., AND SELMAN, A. (1975), A comparison of polynomial time reducibilities, *Theoret. Comput. Sci.* **1**, 103–123.

PAPADIMITRIOU, C., AND YANNAKAKIS, M. (1982), The complexity of facets (and some facets of complexity), *in* "Proc. 14th Ann. ACM Sympos. on Theory of Computing," 255–260.

RACKOFF, C. (1982), Relativized questions involving probabilistic algorithms, *J. Assoc. Comput. Mach.* **29**, 261–268.

SELMAN, A. (1974), On the structure of NP, Notices Amer. Math. Soc. **21**, No. 6, 310.

SIPSER, M. (1982), On relativization and the existence of complete sets, *in* "Proc. 9th Colloq. Automata, Lang. Programming," Lecture Notes in Computer Science Vol. 140, pp. 523–531, Springer-Verlag.

ULLIAN, J. (1967), Partial algorithm problems for context-free languages, *Inform. Contr.* **11**, 80–101.

VALIANT, L. (1976), Relative complexity of checking and evaluating, *Inform. Process.* **5**, 20–23.